```
        && (in >> x)                // got x value
        && (in >> c) && c == ','    // got ','
        && (in >> y)                // got y value
        && (in >> c) && c == ')')   // got ')'
    {
        p.set_to(x, y);
    }
    else {   // input failure
        in.clear(ios::failbit);
    }

    return in;
}
```

---

3. Since the I/O operations have parameters of type **ostream &** and **istream &**, they can be used with **ofstreams** and **ifstreams**. Therefore, we do not need to add any code to our class to read and write **Points** to files.

```
}

Point Point::operator/(int i) const
{
   Point temp;
   temp._x = _x / i;
   temp._y = _y / i;
   return temp;
}

int Point::operator==(const Point &p) const
{
   return (_x == p._x && _y == p._y)?1:0;
}

int Point::operator!=(const Point &p) const
{
   return (_x != p._x || _y != p._y)?1:0;
}

Point &Point::operator+=(const Point &p)
{
   *this = *this + p;
   return *this;
}

Point::Point(int x, int y)
{
   _x = x;
   _y = y;
   other_ctor_count++;
}

Point::Point()
{
   default_ctor_count++;
}


void Point::print_stats(ostream &out)
{
     out << "There were " << default_ctor_count <<
         " calls to the default constructor,\n";
     out << "       and " << other_ctor_count <<
         " calls to the other constructor,\n";
     out << " totalling " << default_ctor_count + other_ctor_count <<
         " all together.\n";
}

ostream & operator<<(ostream &out, const Point &p)
{
     return out << '(' << p.x() << ", " << p.y() << ')';
}

istream & operator>>(istream &in , Point &p)
{
     int x, y;
     char c;

     if ( (in >> c) && c == '('      // got '('
```

Note that the input and output operations can be written in terms of the existing defining operations, so we do not need to add them to the class itself. We have chosen to declare them in the *Point.h* header file to make them available to users of class **Point**. We have also modified the print_stats function to work with an output stream rather than a **FILE *** argument.

The output operator can be written in terms of output of integers and characters. The input operator can read in the x and y values of the point, and it should check to make sure the proper characters surround these values. If it encounters bad input, it sets the failbit in the state of the stream, so that users who later check the state of the stream will be alerted to the error.

FILE: **point.c**

```
#include "Point.h"
#include <stdio.h>

int Point::default_ctor_count = 0;
int Point::other_ctor_count = 0;

void Point::set_to(int x, int y)
{
   _x = x;
   _y = y;
}

Point Point::operator+(const Point &p) const
{
   Point temp;
   temp._x = _x + p._x;
   temp._y = _y + p._y;
   return temp;
}

Point Point::operator-(const Point &p) const
{
   Point temp;
   temp._x = _x - p._x;
   temp._y = _y - p._y;
   return temp;
}

Point operator*(int i, const Point &p)
{
   Point temp;
   temp._x = i * p._x;
   temp._y = i * p._y;
   return temp;
}

Point operator*(const Point &p, int i)
{
   Point temp;
   temp._x = i * p._x;
   temp._y = i * p._y;
   return temp;
```

UNIT 14

Lab Exercises (Answers)

1. The new class **Point**, for exercises 1 and 2 (containing input and output operations) looks like this:

FILE: **Point.h**

```
#include <iostream.h>

class Point {
public:
        Point(int x, int y);
        Point();

        int x() const;
        int y() const;
        void set_to(int x, int y);

        Point operator+(const Point &) const;
        Point operator-(const Point &) const;
        friend Point operator*(int,  const Point &);
        friend Point operator*(const Point &, int);
        Point operator/(int) const;

        int operator==(const Point &) const;
        int operator!=(const Point &) const;

        Point &operator+=(const Point &);

        static void print_stats(ostream &out = cout);

private:
        int _x;
        int _y;

        static int default_ctor_count;
        static int other_ctor_count;
};

// Read & Print points in the format "(x, y)"
ostream & operator<<(ostream &out, const Point &p);
istream & operator>>(istream &in , Point &p);

inline int Point::x() const
{
   return (_x);
}

inline int Point::y() const
{
   return(_y);
}
```

SUMMARY	
DIRECTORY	unit14/point
DECLARATION	Point.h
IMPLEMENTATION	point.c
TEST PROGRAM	point_file.c

FILE: **point_file.c**

```
#include <fstream.h>
#include "Point.h"

int create_file()
{
        ofstream test("point_io.test");

        test << Point(1, 1) << " " << Point(10, 10) << "\n";

        return test.good();
        // file closed upon return from function
}

int read_file()
{
        ifstream test2("point_io.test");
        Point a, b;

        test2 >> a >> b;
        if (test2) {
                cout << a << "+" << b << " is " << a + b << "\n";
                if (a + b != Point(11, 11)) {
                        cerr << "wrong input.\n";
                        return 0;
                }
                return 1;
        }

        cerr << "failed to read input.\n";
        return 0;
}

main(int, char *[])
{
        if(create_file() && read_file())
                cout << "Test successful\n";

        return 0;
}
```

can compile and execute it directly using the commands:

```
$ CC point_in.c point.c -o point_in
$ point_in
```

SUMMARY	
DIRECTORY	unit14/point
DECLARATION	Point.h (modify)
IMPLEMENTATION	point.c (modify)
TEST PROGRAM	point_in.c

FILE: **point_in.c**

```
#include <iostream.h>
#include "Point.h"

main(int, char *[])
{
        Point a, b;

        cout << "Enter point a: ";
        cin >> a;
        if (!cin) {
                cout << "input failure on Point a.\n";
                return 1;
        }

        cout << "Enter point b: ";
        cin >> b;
        if (!cin) {
                cout << "input failure on Point b.\n";
                return 2;
        }

        cout << a << "+" << b << " is " << a + b << "\n";

        return 0;
}
```

3. Do you have to write any additional functions to input or output **Points** to files? Compile and run the test program *point_file.c*.

   You can compile and execute the test program by entering '**make prob3**' or you can compile and execute it directly using the commands:

```
$ CC point_file.c point.c -o point_file
$ point_file
```

UNIT 14

Lab Exercises

1. Change to the *unit14/point* directory. Create an **operator<<** function to output **Points**. Declare the function in *Point.h* and write the implementation code in the file *point.c*. Test the function with the test program *point_out.c*. While you are modifying the files *Point.h* and *point.c*, modify the **print_stats** function to work with an output stream (**ostream**) rather than a **FILE *** argument.

   You can compile and execute the test program by entering '**make prob1**' or you can compile and execute it directly using the commands:

```
$ CC point_out.c point.c -o point_out
$ point_out
```

SUMMARY	
DIRECTORY	unit14/point
DECLARATION	Point.h (modify)
IMPLEMENTATION	point.c (modify)
TEST PROGRAM	point_out.c

FILE: **point_out.c**

```
#include <iostream.h>
#include "Point.h"

main(int, char *[])
{
        Point a (1, 1), b(10, 10);

        cout << "a is " << a << "\n";

        cout << "b is " << b << "\n";

        return 0;
}
```

2. Create an **operator>>** function to input **Points**, and test it with the file *point_in.c*. Note that your input operation should accept as input anything that the output operation prints (so that one could use the output of *point_out.c* as input to *point_in.c*).

   You can compile and execute the test program by entering '**make prob2**' or you

Lab Exercises

Summary

In this unit, we have seen how to:

- Use the C++ Stream I/O facility.
- Work with file streams
- Define I/O operations for a new type.

Summary

In this unit, we have seen how to use this facility to perform I/O on variables of built-in types or classes.

String I/O

```
#include "String.h"
#include <fstream.h>

main(int, char *[])
{
    String firstname, lastname;

    cerr << "Enter your first name, please: ";
    cin >> firstname;

    cerr << "and now your last name: ";
    cin >> lastname;

    cout << "Your name is: "
      << firstname + " " + lastname
      << ".\n";

    ofstream namefile("/tmp/name", ios::out | ios::app);
    namefile << "Processed name: "
         <<  firstname + " " + lastname
         << "\n";

    return 0;
}
```

String I/O

Now that we have defined an input operation and an output operation for Strings, we can input and output Strings.

String input

```
#include "String.h"
#include <stdlib.h>

istream &operator>>(istream &in, String &s)
{
    char nextch;
    int size = 0;  // current size

    // free up old storage, allocate some space

    while(1) {
      in.get(nextch);  // sets nextch
      if (!in || nextch == '\n') {
          s.heap_ptr[size] = '\0';
          return in;
      }
      s.heap_ptr[size++] = nextch;

      // if in need of more storage, re-allocate
    }
}
```

String input

Input is somewhat more complicated. There is no existing input operation to read in a whole line of text, so we have to construct a loop that reads in 1 character at a time (using the "get" operation on iostreams). If it runs out of storage, it allocates more (this code has nothing to do with I/O, so it is not shown here).

We could have used an istream function that reads line of text into a fixed size buffer of characters rather than reading one character at a time, but we would still have to deal with re-allocation of storage if the line were longer than the buffer.

Note that this operation does the two things all input operations must do: it reads the input, and it returns the istream it read from.

String output

```
#include "String.h"

ostream &operator<<(ostream &out, const String &s)
{
    return out << s.heap_ptr;
        // uses operator<<(char*)
}
```

String output

The output operation for `Strings` can easily be written in terms of output operations on simpler types. The member `heap_ptr` of the `String s` has type `char *`, so it will be output using the existing output operation for type `char *`. All output operators are supposed to return the `ostream` they wrote on, so our output operator must then return the `ostream out`. This could be done with a second statement "return out;", or it could be done by simply returning the result of "out << s.data", which must be "out".

This is a typical output operation -- it requires only a line or two of code that simply outputs the data members of the object using simpler output operations.

Adding I/O operations to a new class

```
#include <iostream.h>
class String {
public:
    String();
    String(const char *);
    String(const String &);
    ~String();

    const char *as_char_pointer() const;

    String &operator=(const String &);
    int  length() const;

  // Replacements for read() and print():
  friend ostream &operator<<(ostream &, const String &);
  friend istream &operator>>(istream &, String &);

    const char &operator [](int) const;
    char & operator [](int);
    String substring(int start, int len) const;

    friend int operator==(const String &, const String &);
    friend int operator!=(const String &, const String &);

    static int n_concatenations();
    friend String operator+(const String &, const String &);
private:

    char *heap_ptr;
    static int concat_calls;
};
```

Adding I/O operations to a new class

To change class String to use stream I/O, we need to do two things: First, we must replace the read and print functions with operator>> and operator<<. Second, we must change the other member functions of class string that did I/O so that they use the stream I/O library. Stream I/O and standard I/O should not be used together on the same file descriptor. Since `cerr` and `stderr` both correspond to file descriptor 2, and we may wish to print error messages from our main program, we should convert our String member functions to use `cerr` rather than `stderr`.

Our new input and output operators need not be either friends or members of the stream classes, because they will use only the public interfaces of those classes. Operators must be friends of String if they will need to access the private data. These will.

The member functions that use stdio can be located by removing the include of stdio.h and watching to see which lines give warnings. For example, the line in String::alloc_and_set(char *s) that reports an error if new fails will not get a warning (because the function fprintf has not been declared):

```
fprintf(stderr, "Insufficient storage for string \"%s\"\n", s);
```

It can be replaced by:

```
cerr << "Insufficient storage for string \"" << s << "\"\n";
```

Using File Streams

```
#include <fstream.h>

main(int, char *[])
{
     ifstream in("/tmp/source");
     ofstream out("/tmp/dest");
     int i;
     double d;

     if (in.good() && out.good()) {
       in >> i >> d;
       out << "read integer: " << i
           << " and double: " << d << ".\n";
     }
}
```

Using File Streams

Class ifstream

```
// Simplified class ifstream:

class ifstream : public istream {
public:
    ifstream() ;
    ifstream(const char* name,
         int mode=ios::in,
         int prot=filebuf::openprot) ;
    ifstream(int fd) ;
    ~ifstream() ;

    void open(const char* name,
          int mode=ios::in,
          int prot=filebuf::openprot) ;
    void close() ;
} ;
```

Class ifstream

Class `ifstream` is similar to class `ofstream`. Its default mode is `ios::in`, and it inherits from `istream` instead of `ostream` (and therefore has `operator>>` functions instead of `operator<<` functions).

The stream I/O library also provides a type `fstream`, for input and output to files, but we will not study it in this chapter for two reasons. First, because class `fstream` is defined with multiple inheritance, which we have not studied, and second, because `fstreams` are are used less frequently than `ifstreams` and `ofstreams`. Files are usually opened for both reading and writing when they are being used to store a large amount of data that may need to be updated. It is often easier to update a file by using class that treats a file as an array. Such a class would provide an operator[] that allows access to different records in the file, and might be used like this:

```
file_of_records F(filename);
record i, j;

i = F[3];  // read record #3
F[2]] = j; // write record #2
```

Class ofstream

```
// Simplified class ofstream:

class ofstream : public ostream {
public:
    ofstream() ;
    ofstream(const char* name,
         int mode=ios::out,
         int prot=filebuf::openprot) ;
    ofstream(int fd) ;
    ~ofstream() ;

    void open(const char* name,
          int mode=ios::out,
          int prot=filebuf::openprot) ;
    void close() ;
} ;
```

Class ofstream

Class `ofstream` adds file access functions to the functions of class `ostream`. An `ofstream` can be opened when it is initialized, or it can be opened later with the `open` member function. The arguments to open and the constructor are:

- The `name` gives the name of the file to be opened.
- The `mode` gives the open mode, which can be any of the values given in the enum `open_mode` given in the class `ios`. For output files, the default mode is `ios::out`.
- The `prot` gives the protection mode that will be used for the file if it is created by call to `open`.

When a file stream is created, the constructor automatically creates a buffer that is appropriate for work with files (an object of class `filebuf`). The `filebuf` is used as the argument to the `ostream` constructor. Since this buffer is managed automatically, our code does not need to work with it. Note that the default mode for file creation is defined by a static member of class `filebuf`.

Using class istream

```
#include <iostream.h>

main(int, char *[])
{
    int i;
    char buffer[256];
    double d;

    cin >> i;        // operator>>(int &);
    cin >> buffer;   // operator>>(char *);
    cin >> d;        // operator>>(double &);

  // cin >> i >> buffer >> d; is equivalent to above

    if (cin) {
      cout << "i is: " << i
           << " buffer is: " << buffer
           << " d is: " << d << '\n';
    }
    else {
      cerr << "input unsuccessful.\n";
    }

    return 0;
}
```

Using class istream

The input operators are all written so that they return a reference to the invoking istream. For example, given an integer variable i, the expression:

```
cin >> i
```

returns "cin". That means that we can use this expression anywhere we could use "cin," such as on the left hand side of an input operation:

```
cin >> i >> "\n";
```

Since cin >> i is equivalent to just "cin" (except for the fact that it read in i), the above is equivalent to:

```
cin >> i;
cin >> "\n";
```

class istream

```
// Simplified class istream:

class istream : public ios {
public:
     istream(streambuf*) ;
     virtual  ~istream() ;

     istream& seekg(streampos p) ;
     istream& seekg(streamoff o, seek_dir d) ;
     streampos    tellg() ;

     istream& operator>>(char*);
     istream& operator>>(unsigned char*);
     istream& operator>>(unsigned char& c);
     istream& operator>>(char& c);
     istream& operator>>(short&);
     istream& operator>>(int&);
     istream& operator>>(long&);
     istream& operator>>(unsigned short&);
     istream& operator>>(unsigned int&);
     istream& operator>>(unsigned long&);
     istream& operator>>(float&);
     istream& operator>>(double&);

     istream& get(char* , int lim, char delim='\n');
     istream& getline(char* b, int lim, char delim='\n');
     istream& get(char& c);
     int  get();
     int  peek() ;
     istream& putback(char c);
};
```

class istream

Class istream has operations to input the primitive types using the right shift operator (>>). These operations must use reference parameters for their right hand operands, so that they can change the calling function's variable.

We can also read characters from a stream with the "get" member functions. The >> operators skip over "white space" when reading characters, but the get functions return the spaces, newlines, and tabs they read. The `peek` function returns the next character in the stream, but without removing it from the stream (so a subsequent get() would get the same character). `putback` puts a character back into the input stream. The next input operation will start by reading the character put back.

Using class ostream

```
#include <iostream.h>

main(int, char *[])
{
    cout << 42;
    cout << "hello world\n";
    cout << 3.1415;

    float a = 4.56;
    int b = 7;
    cout << "the sum of " << a << " and " << b
      << " is: " << a + b << ".\n";

    if (cout.good()) {
      cout << "all I/O successful.\n";
    }
    else {
      cerr << "Some I/O operations failed.\n";
    }

    return 0;
}
```

Using class ostream

The output operators are all written so that they return the invoking ostream object. For example, the expression:

```
cout << 42
```

returns cout. Note that cout is returned by reference, not by value, so "cout << 42" refers to the original variable "cout", not a copy of cout. The expression "cout << 42" can be used anywhere the variable "cout" could. For example, we could use "cout << 42" as the left operand of another output operation:

```
cout << 42 << "\n";

// the above is just like:

cout << 42;
cout << "\n";
```

We can also use the operations declared in `ostream's` base class on an `ostream`. For example, we can determine if the ostream is still in the "good" state with the `good` member function.

class ostream

```
// Simplified class ostream:

class ostream : public ios {
public:
     ostream(streambuf*) ;
     virtual  ~ostream();

     ostream& flush() ;
     ostream& seekp(streampos p) ;
     ostream& seekp(streamoff o, seek_dir d) ;
     streampos    tellp();
     ostream& put(char c);

     ostream& operator<<(char c);
     ostream& operator<<(unsigned char c);
     ostream& operator<<(const char*);
     ostream& operator<<(int a);
     ostream& operator<<(long);
     ostream& operator<<(double);
     ostream& operator<<(float);
     ostream& operator<<(unsigned int a);
     ostream& operator<<(unsigned long);
     ostream& operator<<(void*);
     ostream& operator<<(short i);
     ostream& operator<<(unsigned short i) ;
} ;
```

class ostream

Class `ostream` has operations to output the primitive types using the left shift operator (<<). C++ will automatically choose the appropriate << operator for the type of operand used, just as it always uses the types of the operands to select an overloaded operator. Individual characters can be output with either the << operation or the `put` member function.

The `ostream` constructor requires an argument of type `streambuf *`, so every `ostream` must be given a stream buffer when it is created. A stream buffer handles the buffering of data moving to or from a stream. We will not need to work with stream buffers directly, as they are created automatically by the derived class constructors before the `ostream` constructor is called.

Class `ostream` also provides member functions for flushing the output from the buffer associated with the stream, and seeking to a new position in the stream. The types `streampos` and `streamoff` are both typedef'd to `long`, and are used to represent offsets within a stream.

Class ios

```
// Simplified class ios:

class ios {
public:
    // state of stream:
    int  eof();
    int  fail();
    int  bad();
    int  good();

    enum io_state{ goodbit=0, eofbit=1, failbit=2, badbit=4 };
    int  rdstate();
    void clear(int i =0) ;

    operator void*();
    int  operator!();

    // mode of stream
    enum open_mode    { in=1, out=2, ate=4, app=010,
                trunc=020, nocreate=040, noreplace=0100} ;
    enum seek_dir{ beg=0, cur=1, end=2 } ;

    // other functions involving access to buffer
};
```

Class ios

Class `ios` declares functions for determining the state of a stream. The `eof` function can be used to determine if a stream has reached the end of a file. `bad` will return true if an invalid operation has been performed (e.g., seeking past the end of a file). If an i/o operation was unsuccessful or illegal, the `fail` function will return true. `good` will return true if nothing is wrong with the stream (i.e., if none of the previous three functions would return true).

The `clear` function can be used to set the state of the stream to one of the states given by the enumerated type `io_state`. `rdstate` returns the state of the stream (which could also be determined by calling the four functions listed above).

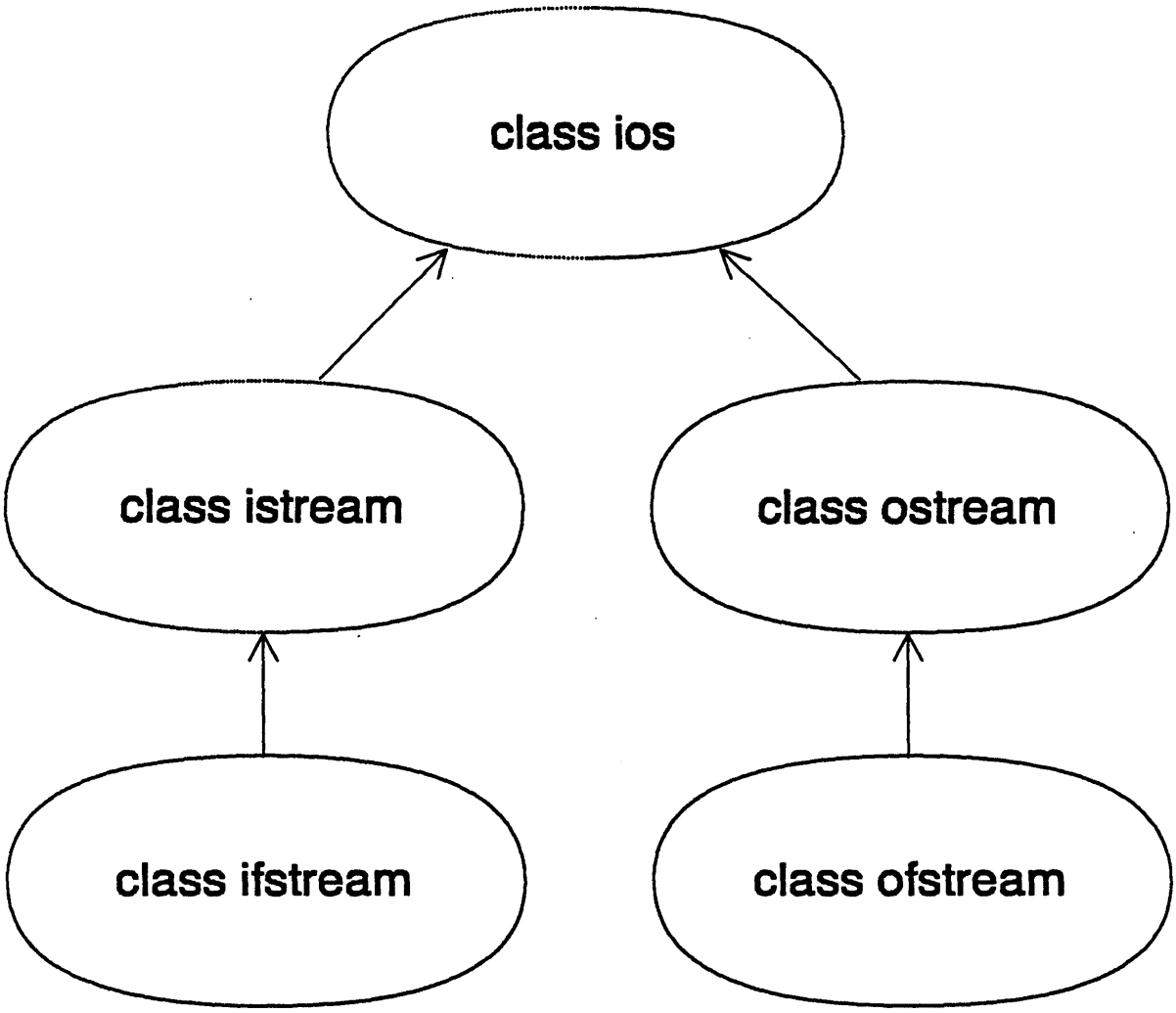
`operator!` and `operator void*` allow convenient access to the state of the stream. The first will return a non-zero integer if the stream is *not* good, the second will return a non-null pointer if the stream's state *is* good. These operators allow the use of stream type values within an **if** statement, where an arithmetic or pointer type value is required:

```
if (cout)     // true if the stream "cout" is good

if ( !cout )  // true if the stream "cout" is not good
```

Class `ios` also declares enumerated types describing the modes that can be used when a stream is opened (`open_mode`), and the direction of a seek operation (`seek_dir`).

I/O Stream Library Classes



I/O Stream Library Classes

Class `ios` contains the features common to both input and output streams: information about the state of the stream, and modes that can be used when opening a new stream.

Class `ostream` defines the output operations common to all output streams. It is used as a base for different output stream classes, such as `ofstream`. Class `ofstream` (output file stream) is a class for output streams that write to files. It adds operations for opening and closing files to the operations it inherits from `ostream`.

Class `istream` is the base class for input streams. It defines the input operations. Class `ifstream` (input file stream) adds operations for opening and closing files to the operations it inherits from `istream`.

Streams

Stream I/O library

- Type-safe
- Extensible
- output to "output stream" object:

```
cout << "Exit program? ";
cerr << "Error encountered\n";
```

- input from "input stream" object:

```
char answer[10];
cin >> answer;
```

Streams

The left shift operator (<<) is used to send output to an output stream, and the right shift (>>) to get input from an input stream. When they are used in this context, << is sometimes called the output operator, and >> the input operator. The library defines the streams `cin`, (an input stream corresponding to the standard input), `cout` (an output stream corresponding to the standard output), and `cerr` (an output stream corresponding to the standard error).

There are three major differences between the stream I/O library and the standard I/O library:

When you use printf or scanf, you must specify the types of the arguments in the format string, and there is no error checking. You could accidentally use the format "%f" with an integer without getting any warning from the compiler. The normal C++ rules for selecting an overloaded operator will automatically select the correct output operation for the type of variable being printed or read in.

It is easy to extend the stream I/O library to include new types. Extending printf or scanf to work with new types is much harder. In fact, it is so hard that most programmers just create other functions to do I/O on new types. Code written in this fashion uses printf for char, int, or float variables, a function like "print_complex" for complex numbers, and a function like "print_foo" for some other type foo.

The stream I/O library also has new mechanisms for working with formatted input or output, but these are outside the scope of this course. The original formatting mechanism is described in Bjarne Stroustrup's book, "The C++ Programming language." Other mechanisms have been added in later versions of the library, and are documented in the release notes.

Objectives

At the end of this unit we will be able to:

- Use the C++ Stream I/O library:
  - — input data
  - — output data
  - — open and close files
  - — define I/O operations for new types

CONTENTS

Unit 14 - IO Streams

Unit 14

Object-Oriented Programming in C++

IO Streams