Input/Output

Stream I/O

- Stream – Sequence of bytes that may be input to, or output from, a program
- Three standard streams in C++
  1. `cin` – Standard input stream (`stdin`)
  2. `cout` – Standard output stream (`stdout`)
  3. `cerr` – Standard error stream (`stderr`)
- `cin` and `cout` are buffered streams while `cerr` is unbuffered

The << and >> operators

- The `cin >>` and `cout <<` operators are overloaded >> and << operators, applied to members `cin` and `cout` of the classes `istream` and `ostream`, defined in `iostream.h`
- Why can the operators be chained together (like `cout << x << y;`)?
  - The operator << binds left to right
  - The above expression can be fully parenthesized as

    `( cout << x ) << y;`
  - Furthermore, each << operation returns a value that is identical to its first argument
  - Since `ostreams` are fairly sizable objects, the `operator<<` functions take an `ostream&` and return it
  - The definition of the operator inside the class is

```
ostream& operator<< ( ostream&, char* );
ostream& operator<< ( ostream&, int );
```

- Defining << for user-defined classes (example `complex`)

```
class complex
{
    double re, im;
    public:
        complex ( double, double );
        friend ostream& operator<< ( ostream&, const complex& );
        friend istream& operator>> ( istream&, complex& );
};

complex::complex ( double r = 0.0, double i = 0.0 )
{
    re = r;
    im = i;
}

ostream& operator<< ( ostream& s, const complex& c )
{
    return ( s << c.re << " + i" << c.im );
}
```

```
istream& operator>> ( istream& s, complex& c )
{
    s >> c.re;
    char ch;
    s >> ch;
    if ( ch != '+' )
        cout << "+ expected" << endl;
    s >> ch;
    if ( ch != 'i' )
        cout << "i expected" << endl;
    s >> c.im;
    return s;
}

main()
{
    complex c ( 5.0, 3.0 );
    cout << "The complex number is: " << c << endl;
}
```

I/O of characters and strings

- We can put a single character on the output stream by using the function

  `ostream::put ( char )`

- The function is called as

  `cout.put ( ch )`

- A character is input by

  `cin.get ( ch )`

  get reads a whitespace character rather than skipping over it

- You can put the character back on the input stream by

  `cin.putback ( ch )`

- You can examine a character without removing it from input stream by

  `cin.peek ( ch )`

- Two functions for line-based input

  ```
  istream& istream::getline ( char * buffer, int bufsize, char terminator = '\n');
  istream& istream::get ( char * buffer, int bufsize, char terminator = '\n');
  ```

  - `cin.getline ( line, 80 );` reads a line
  - `cin.get ( word, 20, ' ' );` reads in a word because the termination character is changed to a space
  - The termination character is left as the next character on the input stream when calling get, whereas getline extracts the termination character and throws it away

  - Both functions automatically place a null string terminator in the buffer, and stop reading when the buffer is full
  - You can call `istream::gcount()` immediately after any of these two functions to get the number of characters transferred into the buffer

Formatted I/O

- Formatting (binary to ASCII and the other way round) is automatically performed by the class `ios`, the base class for both `istream` and `ostream`
- Formatting is governed by a format state in the `ios` class
- The format state contains
  - Field width
  - Fill character
  - Field alignment
  - Integer base (decimal, hex, octal)
  - Floating point format (fixed, scientific, general)
  - Whether to show a + sign, trailing decimal point and zeroes, or the integer base
  - Uppercase or lowercase letter usage for E, 0X, hex digits
- To manipulate these states, you need to `#include <iomanip.h>` in your function
- Each of the states can be changed in two ways
  1. With an `ios` function manipulator
     - The `ios` function manipulators return the old values of the manipulated parameter, making it convenient to restore the old values if needed
  2. With a *manipulator*
- To set the fill character to '0' and the width to 10, you can use

```
cout.fill ( '0' );
cout.width ( 10 );
```

  or

```
cout << setfill ( '0' ) << setw ( 10 );
```

- Setting field width
  - For output, field width denotes the minimum number of characters printed for a field
  - For input, width is meaningful only if reading a string
- Except for field width, the members of format state are implemented with bits or groups of bits
  - Bits can be turned on with the member function `ios::setf` or a manipulator `setiosflags`
  - Bits can be turned off with the member function `ios::unsetf` or a manipulator `resetiosflags`
  - To show a leading + sign for positive numbers, you can use either of the following:

```
cout.setf ( ios::showpos );
cout << setiosflags ( ios::showpos );
```

  - The flag can be turned off by

    ```
    cout.unsetf ( ios::showpos );
    cout << resetiosflags ( ios::showpos );
    ```

  - For more than two choices (such as decimal, hex, or octal base), you need to make two selections

    ```
    cout.setf ( ios::hex, ios::basefield );
    ```

- The available flags are:

ios flag	Field group	Meaning
`left`	`adjustfield`	Left alignment
`right`	`adjustfield`	Right alignment
`internal`	`adjustfield`	Sign left, remainder
`dec`	`basefield`	Decimal base
`hex`	`basefield`	Hex base
`oct`	`basefield`	Octal base
`showbase`		Show integer base
`showpos`		Show + sign
`uppercase`		Uppercase E, X, and A...F
`fixed`	`floatfield`	Fixed floating-point
`scientific`	`floatfield`	Scientific floating-point
`showpoint`		Show trailing decimal point

- Illustrating different features

```
#include <iostream.h>
#include <iomanip.h>

main()
{
    cout << setfill( '0' ) << setw ( 10 );
    cout << 42.84 << endl;
    cout.setf ( ios::left, ios::adjustfield );
    cout << setfill( '*' ) << setw ( 20 ) << "Hello World!" << endl;
    cout.setf ( ios::right, ios::adjustfield );
    cout << setw ( 20 ) << "Hello World!" << endl;
    cout.setf ( ios::scientific, ios::floatfield);
    cout.setf ( ios::showpoint | ios::showpos );
    cout << setprecision ( 10 ) << 123.45678 << endl;
    cout.unsetf ( ios::floatfield);
    cout << setprecision ( 10 ) << 123.45678 << endl;

    int n = 12;
    cout.unsetf ( ios::showpos );
    cout << hex << n << ' ' << oct << n << ' ' << dec << n << endl;
}
```

- Available manipulators

Manipulator	ios member function
setfill	fill
setw	width
setprecision	precision
setiosflags	setf
resetiosflags	unsetf
hex	
oct	
dec	

- The only saving grace compared to C I/O handling is that C++ completely avoids the mismatch between the format string and the data type

Using files for I/O

- You must include the header file `fstream.h`
- To open a stream attached to a file

```
ifstream is ( "input.txt", ios::in );
ofstream os ( "output.txt", ios::out );
```

- The classes `ifstream` and `ofstream` are derived from `istream` and `ostream`
  - An `[io]fstream` object is an `[io]stream` object with added capabilities
  - You can use all the `[io]stream` functions (<<, `get`, `putback`) with `[io]fstream` objects
- Example to read something from terminal and put it to a file

```
#include <iostream.h>
#include <fstream.h>

main()
{
    char ch, line[80];

    // Open output stream for file outfile

    ofstream os ( "outfile", ios::out );

    while ( !( ch = cin.eof() ) )
    {
        cin.getline ( line, sizeof(line) );
        os << line << endl;
    }

    // Close the output stream

    os.close();
}
```

  - If a file cannot be opened, the `[io]fstream` variable gets 0 and can be checked

```
if ( ! os )
    cerr << "Error opening file" << endl;
```

Error states

- All I/O streams have a state associated with them from the set

  `{good, end-of-file, fail, bad}`

- Fail state
  - Occurs because of a logical error, such as trying to read an integer when the next character is a letter
- Bad state
  - Occurs because of a physical error, such as a hardware failure or memory exhaustion
- The stream state is reported by four access functions
  1. `good()`
  2. `eof()`
  3. `bad()`
  4. `fail()`
- `fail()` is true if the stream is in "fail" or "bad" state
- `eof()` is true if the end of file is reached and the stream is not in a failed state
- The states can be tested with the `rdstate` member function

  ```
  if ( s.rdstate() & ios::failbit )
  ```

- The stream can be reset to "good" state with

  ```
  s.clear();
  ```

  or to "fail" with

  ```
  s.clear ( ios::failbit | s.rdstate() );
  ```

  - You may need to clear the state if you want to continue reading after the state has been set to "bad" or "fail"
  - You may need to set the state to "bad" to report problems in the user-defined `operator>>`

Attaching streams to strings

- Equivalent to `sscanf`
- Achieved by the string stream package

```
#include <iostream.h>
#include <strstream.h>

main()
{
    char line[80];
    int i;
    double d;
    istrstream si ( line, sizeof ( line ) );
```

```
    while ( cin.getline ( line, sizeof ( line ) ) )
    {
        si >> i >> d;
        si.seekg ( ios::beg );  //reset to start
        cout << i << ' ' << d << endl;
    }
}
```

- The output stream is declared with

```
ostrstream so ( line, sizeof ( line), ios::out );
```

  and is reset with

```
so.seekp ( ios::beg );
```