C++ Stream I/O

Streams

- A stream is a sequence of bytes
  - Input: flow from a device to memory
  - Output: flow from memory to a device

Stream I/O Classes and Objects

- cin: an instance of istream tied to stdin
- cout: an instance of ostream tied to stdout
- cerr, clog: instances of ostream tied to std error. Outputs to cerr are unbuffered, to clog is buffered
Stream Output

• Stream-insertion operator `<<` is overloaded to output data items of built-in types, strings, and pointer values

```cpp
cout << "hello\n";
cout << "hello" << endl;
```

• The stream manipulator `endl` flushes the output buffer, while `\n` does not.

Stream Output

```cpp
cout << "abc" << (20+30) << endl;
```

• Cascading is allowed because `<<` returns a reference to `cout`

```cpp
cout.put('A'); cout.put(65); // both output 'A'
```

• The `put` member function outputs one character

Stream Input

• The stream extraction operator `>>` returns 0 if end-of-file is encountered; otherwise returns a reference to the object that received the message (e.g. `cin`)
  - Cascading is allowed
  - end-of-file: `<ctrl>-d` (UNIX, Mac), `<ctrl>-z` (PC)
**istream Member Functions**

- `get()`, `getline()`
- `ignore()`: skips over a designed number of characters or delimiter
- `putback()`: places the previous character obtained by `get()` back onto that stream
- `peek()`: returns next character from an input, but doesn't remove it from the stream

**Unformatted I/O**

- Inputs or outputs some number of bytes to or from a character array in memory

```cpp
void main(){
    char buffer[80];
    cout<<"Enter: \n";
    cin.read(buffer,20);
    cout<<"Entered was: \n";
    cout.write(buffer,
        cin.gcount());
    cout<<endl;
}
```

Enter:
Using the read, write, gcount
Entered was:
Using the read, write

**Integral Stream Base**

- `dec(base 10)`, `oct(base 8)`, `hex(base 16)`, `setbase()` (parameter 10, 8, or 16)

```cpp
#include<iomanip>
void main(){
    int n;
    cout<<"Enter a num: \n";
    cin>>n;
    cout<<hex<<n<<"\n";
    cout<<oct<<n<<"\n";
    cout<<dec<<n;
    cout<<setbase(10)<<n;
    cout<<endl;
}
```

Enter a num:
20
14 24 20 20
Floating-Point Precision

- Use member function `precision()`, or stream manipulator `setprecision`

```cpp
#include<iomanip>

void main(){
    double root2=sqrt(2.0);
    for(int i=1;i<5;i++){
        cout.precision(i);
        cout << root2 << " ";
    }
    cout << endl;
    for(int i=5; i<7; i++)
        cout<<setprecision(i) << root2 << " ";
};
```

User-Defined Manipulators

- User may create their own stream manipulators

```cpp
#include <iostream>

ostream& bell(ostream& output){return output<<'a';}
ostream& tab(ostream& output){return output<<'t';}
ostream& endLine(ostream& output){
    return output<<'n'<<flush;
}

void main(){
    cout << 'a'<<tab<<'b'<<bell<<endLine;
}
```

Stream Format States

- Trailing zeros

```cpp
cout<<9.00;  //output 9
cout.setf(ios::showpoint);
cout<<9.00;  //output 9.00
```

- Justification

```cpp
int x = 12345;
cout<<setw(10)<<x;  //output 12345
cout.setf(ios::left, ios::adjustfield);
cout<<setw(10)<<x;  //output | 12345 |
cout<<setw(10)<<setfill('*')<<x;  //output |12345*****|
```
Stream Format States

• Integral Stream Base

```cpp
int x = 100;
cout << setiosflags(ios::showbase) << x // output 100
     << oct << x // output 0144
     << hex << x; // output 0x64
```

Stream Format States

• Floating-Point Numbers

```cpp
double x = 0.001234567, y = 1.946e9;
cout << x << y; // output 0.001234567 1.946e+009
cout.setf(ios::scientific, ios::floatfield);
cout << x << y; // output 1.234567e-003 1.946000e+009
cout.unsetf(ios::scientific);
cout << x << y; // output 0.001235 1946000000.000000
```

Stream Error States

• Floating-Point Numbers

```cpp
int x;
cout << "Before a bad input operation" << cin.rdstate() // output 0
    << cin.eof() // output 0
    << cin.fail() // last i/o failed - output 0
    << cin.bad() // invalid operation - output 0
    << cin.good(); // output 1
cin >> x; // input a character 'A'
cout << "After a bad input operation" << cin.rdstate() // output 2
    << cin.eof() // output 0
    << cin.fail() // output 0
    << cin.bad() // output 0
    << cin.good(); // output 0
```
**File Processing**

**Introduction**

- Files are used for permanent retention of large amount of data
  - Storage of data in variables is temporary
- Files are stored on secondary storage devices
  - Magnetic disks, optical disks, tapes

**Data Hierarchy**

- Bit: manipulated by computer circuitry
  - Binary digit: 0 or 1
- Byte: commonly composed of 8 bits
  - Decimal digits, letters, special symbols
- Field: a group of bytes that conveys meaning
- Record: composed of related fields
- File: a group of related records
- Database: a group of related files
Files and Streams

- C++ views files as a sequence of bytes
  - No concept of record
- When a file is opened, an object is created and a stream is associated with it
- To process file in C++, `<iostream>`, `<fstream>` must be included.

Creating a Sequential File

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

void main() {
    ofstream clientF("clients.dat", ios::out);
    if (!clientF) {
        cerr<<"File could not be opened"<<endl;
        exit(1);
    }
    int account; char name[30]; double balance;
    while (cin>>account>>name>>balance)
        clientF<<account<<name<<balance<<'\n';
}
```

- Files are opened by creating objects of `ifstream`, `ofstream`, or `fstream`
- Explicitly close file: `clientF.close();`

Reading From a Sequential File

```cpp
#include<iostream>
#include<fstream>
#include<iomanip>

void main() {
    ifstream clientF("clients.dat", ios::in);
    if (!clientF) {
        cerr<<"File could not be opened"<<endl;
        exit(1);
    }
    int account; char name[30]; double balance;
    while (clientF>>account>>name>>balance)
        cout<<setiosflags(ios::left)<<setw(10)<<account<<setw(13)<<name<<setw(7)<<setprecision(2)<<resetiosflags(ios::left)<<balance<<'\n';
}
```
Reposition File Position Pointer

- Both `istream` and `ostream` provide member function for repositioning
  - `seekg()`, `seekp()`

```cpp
istream fileObj("abc.dat");
fileObj.seekg(n, ios::beg);
//position to the n'th byte of fileObj
fileObj.seekg(n, ios::cur);
//position n bytes forward from current position
fileObj.seekg(n, ios::end);
//position n bytes back from end of fileObj
long loc = fileObj.tellg();
```

Updating Sequential Files

- There is no easy way!
  - Copy all records before White to new file `nf`
  - Append Worthington record to `nf`
  - Append all records after White to `nf`

- If record were rewritten beginning at the same location

<table>
<thead>
<tr>
<th>White: 300.00</th>
<th>Worthington: 300.00</th>
<th>James: 0.20</th>
</tr>
</thead>
<tbody>
<tr>
<td>300 White 0.00</td>
<td>300 Worthington 0.00</td>
<td>400 James 0.20</td>
</tr>
</tbody>
</table>

- Awkward method
  - Copy all records before White to new file `nf`
  - Append Worthington record to `nf`
  - Append all records after White to `nf`

Random-Access Files

- Sequential access files are inappropriate for instant-access applications
  - Airline reservation systems, banking systems
  - Require individual records to be accessed directly without searching through others

- C++ does not impose structure on file
  - Application must create random-access files
Create Random Access Files

- Require records have same fixed length
  - Program can calculate the exact location of any record relative to the beginning of files
  - `<<` should not be used

```cpp
int number;  //number is a 4-byte integer
outFile << number; //print 1 to 11 digits, each digit requires 1 byte
```

- Use `write()` instead

```cpp
outFile.write(reinterpret_cast<const char*>(&number), sizeof(number)); //always write 4 bytes
```

Create Random Access Files

- Require records have same fixed length

```cpp
struct clientD {
  int acct; char la[15]; char fi[10]; double ba;
};
void main(){
  ofstream outF("credit.dat", ios::binary);
  if (!outF) …
  clientD client = {0, "", "", 0.0};
  for (int i=0; i<100; i++)
    outF.write(reinterpret_cast<const char*>(&client), sizeof(client));
}
```

Write to Random Access Files

- Records have fixed length `sizeof(client)`

```cpp
void main(){
  ofstream outF("credit.dat", ios::binary);
  if (!outF) …
  clientD client;
  cin >> client.acct; //1..100
  while (client.acct>0 && client.acct<=100) {
    cin>>client.fi>>client.la>>client.ba;
    //Put file position pointer for object outF
    //to the right byte location
    outF.seekp((client.acct-1)*sizeof(client));
    outF.write(reinterpret_cast<const char*>(&client), sizeof(client));
    cin>>client.acct;
  }
```
void main()
{
    ifstream inF("credit.dat", ios::in);
    if (!inF) …
    clientD client;
    inF.read(reinterpret_cast<const char*>(&client), sizeof(client));
    while (inF && !inF.eof()) {
        if (client.acct != 0)
            outputLine(cout, client);
        inF.read(reinterpret_cast<const char*>(&client), sizeof(client));
    }
}

void outputLine(ostream &output, const clientD &c){
    output << setiosflags(ios::left) << setw(10) << c.acct << setw(16) << c.la
    << setw(11) << c.fi << setw(10) << setprecision(2) << resetiosflags(ios::left)
    << c.balance<<endl;
}

Summary

• C++ imposes no structure on a file. It views each file as a sequential stream of bytes.
• Files are opened by instantiating objects of stream classes ifstream, ofstream, and fstream.
• Streams provide communication channels between files and programs.
• A convenient way to implement random-access files is by using only fixed-length records.