“File” Input and Output

```
#include <iostream.h>
...
int XX; char YY; float ZZ;
cin >> XX >> YY >> ZZ;
cout << XX << YY << ZZ;
---Run-------------
% cinout1
321 $ 12.5
321$12.5
% cinout1
12.5$321
12.5
% cinout1
12 . 5$321
12.5
```
Slide 2

- **class** `istream`: for input from a stream of bytes.
- **class** `ostream`: for output to a stream of bytes.
- `cin`: Predefined object of type `istream`.
- `cout`: Predefined object of type `ostream`.
- `>>, <<`: extraction “get from”, insertion “put to”
  Overloaded member operator functions belonging to the `istream, ostream` classes.
What’s an **Overloaded Operator**?

It is an operator that operates **DIFFERENTLY** according to **DIFFERENT** types of operands. Example:

+ is built-in to C++:

```cpp
int X;
float Z;
X = X + 39;  // uses a binary int
Z = Z + 3.14159;  // uses an IEEE float
```

**add**

**machine instruction**

C++ library writers and other programmers can code their own overloaded operator functions.
cin >> XX; cin >> YY; cin >> ZZ;

The C++ compiler chooses DIFFERENT member functions of class istream to call according to the DIFFERENT types of XX, YY and ZZ

---interface-------

```cpp
class istream {
public:
    istream& operator>>( int& n ); //read into n
    istream& operator>>( char& n ); //read into n
    istream& operator>>( float& n ); //read into n
    //.. }
};
```

---implementation-------

```cpp
istream& istream::operator>>( int &n )
{
    // skip whitespace, scan digit characters
    // until a non-digit stops the scan.
    n = ..integer converted from the digit chars..;
    return *this;
}
```

(operator>>(..) returns a reference to the istream that >> was called with so that we can code:

( (cin >> XX) >> YY ) >> ZZ ;
(cin >> XX) returns the value cin)
What is a “digit”? Depends on the base (radix): Ten, Eight(0-7) or Sixteen(0-9,A-F,a-f).

The FORMAT STATE of the stream controls what base to use: (Strou. 21.4) To set input conversion to hex (required for project 3):

MyInput.setf( ios::hex, ios::basefield );
// alternative:
MyInput << hex; //"more elegant" manipulator
Stream input/output is from/to a **stream** of **bytes**. Stream means the bytes (appear)/(are consumed) sequentially.

Possible streams:

- Terminal input/output.
- File: Named and persistent data store. Data doesn’t disappear when program execution ends.
- (Network or interprocess communication connection: socket or pipe.)

---

Using a C++ stream to access a File.

- Use the pre-defined streams `cin/cout`, but OUTSIDE of the C++ program, use the Unix shell to **redirect** standard input/output to a named file.
  
  (Way for Project 2.)
  
  % cinout1 < datafile.in > datafile.out

- Define another stream object of the class `ifstream/ofstream` or `fstream` which implements a stream connected to a named file.
  
  (REQUIRED for Project 3.)
```cpp
#include <iostream.h>
#include <fstream.h>
int main(int argc, char *argv[]) {
    ifstream Myinput( pointer to char array );
    Defines local scope automatic variable Myinput and calls
    a constructor function to interact with the Operating
    System. The array should contain the file name.
    if( Myinput == 0 )
    { cerr << "Cannot open file\n"; exit (1); }
    while( Myinput >> X )
    { // process current input.. }
```
class ios {
public: //..
    bool good() const; //next op might succeed
    bool eof() const; //end of input SEEN
    bool fail() const; //NEXT OP WILL FAIL
    bool bad() const; //maybe lost data
    operator void*() const; //non-zero if !fail()
    bool operator!() const {return( fail());}
    //..};

Statements if( Myinput == 0 ) and
while(Myinput >> XX ) call the
operator void*(){} .. } member function, to test for failures.
Where can you get the file name?

"Toy way:" ifstream Toy("hardcodedname.data");
COMMAND LINE ARGUMENTS:
% cinout1  BLAH_ONE  ARGUTHREE    --simulate
int main( int argc, char *argv[] ) or
int main( int argc, char **argv )

argc=Number of arguments INCLUDING the name
under which the program was run (here, cinout1)
argv=ptr to char array containing argc pointers to the
arguments as C strings.

That is one way to get RUN TIME parameters. See
next slide for a few others.
• Prompt user and read from terminal.

• Open and read an application configuration file.

• Environment: Array similar to command line array but null terminated and accessed through global static

  extern char **environ;

  (The environment is maintained by the shell and can be modified with csh `setenv` VAR or (ba)sh
  `export` VAR= commands.)

• System database access functions. (Windows Registry, Unix NIS, many others).

---

Streams that access files\(^a\) have a **File Position Indicator**.

```
0 1 2 3 4
```

File Position Indicator (FP)

* Gives the number of the byte to be read next.
* Set to zero when file is opened.
* Increases as bytes are read from file.
* Value provided by `tellg()` function. ("tellg" denotes "tell get".)

\(^a\) (Also stringstreams.)
Example:

```cpp
    ifstream infile("input.dat");
    
    long pos = infile.tellg();
```

**Function seekg:**

- To move around in an input file.
- Uses two arguments:
  - First: Type `long` specifies offset (may be negative).
  - Second: Gives seek direction.

---

- Possible values for seek direction: `ios::beg` (the default), `ios::cur` and `ios::end`.

**Example:** Suppose an input file (corresponding to variable `infile`) is 300 bytes long. (These bytes are numbered 0 through 299.) Initially, FP is zero.

```cpp
    infile.seekg(5); // Sets FP = 5.
    infile.seekg(7,ios::cur); // FP = 12.
    infile.seekg(-3,ios::cur); // FP = 9.
    infile.seekg(0,ios::end); // FP = 299.
    infile.seekg(0); // FP = 0.
```

(b) Output files:
• FP gives the number of the byte to be written next.
• FP set to zero when file is opened with mode `ios::out`.
• FP increases as bytes are added to file.
• Value of FP provided by `tellp()`.
  ("tellp" denotes "tell put".)

**Example:**

```cpp
doctrine ofstream outfile("output.dat");
    .
long pos = outfile.tellp();
```

---

**Function `seekp`:**

• To move around in an output file.
• Uses two arguments similar to `seekg()`.
• Previously written bytes can be changed.

**Another mode for file opening:** Mode `ios::ate`

• Can be used with input or output files.
• Opens the file and sets FPP to the last byte for input file and one beyond the last byte for output files.
Example:

```cpp
ifstream infile ("in.dat", ios::in |
    ios::ate);
```

The mode `ios::app` (for an output file) is equivalent to

```cpp
ios::noreplace | ios::ate
```

Moving outside file boundary:

- `seekg()` and `seekp()` don’t check whether specified move is within the file.
- For illegal moves, effect is implementation dependent.

For the systems in ECL:

- `seekg()` allows any offset value. Trying to read from a non-existent position produces EOF.
- `seekp()` allows “forward jumps”. Positions where nothing was written contain ’\0’.