Files CSI333

Slide 1

"File:" Persistant Storage;  
(Unix:) Data transfer channel.

#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
• 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.

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 persistant data store. Data
doesn’t disappear when program execution ends.
• (Network or interprocess communication
  connection: socket or pipe.)
iostream CONSTRUCTORS for constructing a stream that will “be connected to” a NAMED UNIX FILE:
#include <iostream.h>
#include <fstream.h> ...
ifstream Toy(file name C-string);
Inputting from the named file:
if( Toy == false )
{ // next operation on Toy will fail.
   cerr << "Cannot open" <<
        " file name C-string" << endl;
   exit( 1 );
}
Toy >> XX >> YY >> ZZ;

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.
Command line arg. processing logic:

```cpp
// global variables or options structure:
bool doptgiven, soptgiven; char *filename;
...
int main(int argc, char *argv[]){ ... 
    for( i = 1; i < argc; i++ ) {
        Examine argv[i] and record information about
        the given option. Eg, if(strcmp(argv[i],"-d")
        { ... If the option has an argument,
        do i++ and save the argument; }
        } ... 
    if(doptgiven){ ... }
    if(soptgiven){ ... } 
(Proj. 5 uses an indefinite # of args.)
That is one way to get RUN TIME parameters. See
next slide for a few others.
```
Slide 8

- 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
  
  ```c
  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).

Slide 9

“Formatted” versus “Unformatted” Files (all files are formatted somehow)

Unix has NO BUILT IN INTERPRETATIONS of application file contents: Every file is a stream of 8 bit bytes.

“Formatted, ASCII, text” file usually means a file that (should) contain only bytes in the (printable ASCII) $32_{10}$ to $126_{10}$ plus whitespace (8-13) range.

The formatted stream “get from” (>>) and “put to” (<<) operators convert between conventional printed data value forms and the representations in C++ variables as bits.
>> and << are **type-safe**: The type of C++ variable they are applied to controls what they do, so

- Input format errors are detected; target variable is unchanged when there's a format error.

- Bits in each output variable are interpreted according to the variable's C++ type.

- Type mismatch is avoided.

Format options can be set for individual streams using members of class `ios` (Strou. 21.4).

---

Situations where formatted I/O is not desirable:

1. Complicated, application specific rules for analyzing input: Writing a C++ preprocessor or compiler, for example. Input should be read character by character (or line by line) and processed by a scanner.

2. Bits in the file should be identical with bits processed in C++ memory.

3. Stream or file structure specified as a byte sequence: Network packets or messages, the JAVA VIRTUAL MACHINE .class files FOR PROJECT 5, or object/executable program files.
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.)

\[
\% \text{cinout1} < \text{datafile.in} > \text{datafile.out}
\]

- Define another stream object of the class ifstream/ofstream or fstream which implements a stream connected to a named file. (This or low-level POSIX open/read/write is REQUIRED for Project 5.)

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

int main(int argc, char *argv[]) {

    ifstream Myinput( "pointer to char array" );
    char inbuf[BUFSIZE];
    while( Myinput.get( inbuf[i] ) )
        { i++; // process current input.. }
```

\[
\text{Type of inbuf[i] is char}
\]
Example: char. by char. copy pgm. from Stroustrup:

```cpp
int main()
{ char c; while( cin.get(c) ) cout.put(c); }
```

Demo:

```
$ copy > out.dat
HI
There
$
```

(After There I pressed enter and Control-d)

```
$ copy < out.dat
HI
There
$
```

---

How to read a 256 MIPS words:

```cpp
cost int NWORDS = 256;
long unsigned int pMem =
    new long unsigned int[ NWORDS ];
MyF.read( pMem,
    NWORDS*sizeof(long unsigned int) );
if( MyF.gcount() !=
    NWORDS*sizeof(long unsigned int) )
{ // something’s wrong! }
```

`istream::read( pointer, count )` takes a pointer to the destination and a count of the maximum number of bytes to read. There BETTER be at least count bytes available where pointer points!
How to write a 256 MIPS words:

```c
const int NWORDS = 256;
long unsigned int *pMem =
    new long unsigned int[NWORDS];
MyOutF.write( pMem,
    NWORDS*sizeof(long unsigned int) );
```

ostream::write( pointer, count ) takes a pointer to the source array and a count of the number of bytes to copy out and write. It is the programmer’s responsibility to make sure the output is meaningful.

---

Functions for line-by-line input (illustrated on MyF)
```c
char inbuf[BUF SIZE];
MyF.getline( inbuf, BUF SIZE );
MyF.getline( inbuf, BUF SIZE, 'Z' );
    // Strange line terminator Z
MyF.get( inbuf, BUF SIZE );
    // LEAVES \n in the stream
cout << "Just read " << MyF.gcount()
<< " chars!" << endl;
```
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.
Streams that access files have a **File Position Indicator**.

* 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.

Slide 21

- 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::cur); // 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
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’.
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;
Z = Z + 3.14159;
uses a binary int uses an IEEE float
add add.s
machine instruction machine instruction
```

C++ library writers and other programmers can code their own overloaded operator functions.

---

```cpp
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

---

```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;
}
```

(\texttt{operator\textgreater\textgreater(\ldots\rangle}) returns a reference to the \texttt{istream} that \texttt{\textgreater\textgreater} was called with so that we can code:

\begin{verbatim}
( (cin \textgreater\textgreater XX) \textgreater\textgreater YY ) \textgreater\textgreater ZZ ;
\end{verbatim}

(\texttt{cin \textgreater\textgreater XX}) returns the value \texttt{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):

```cpp
MyInput.setf( ios::hex, ios::basefield );
// alternative:
MyInput \textless\textless hex; //"more elegant" manipulator
```