Chapter 6

I/O Streams as an Introduction to Objects and Classes

Prof. Chien-Nan (Jimmy) Liu
Dept. of Electrical Engineering
National Chiao-Tung Univ.

Tel: (03)5712121 ext:31211
E-mail: jimmyliu@nctu.edu.tw
http://mseda.ee.nctu.edu.tw/jimmyliu

Overview

Streams and Basic File I/O

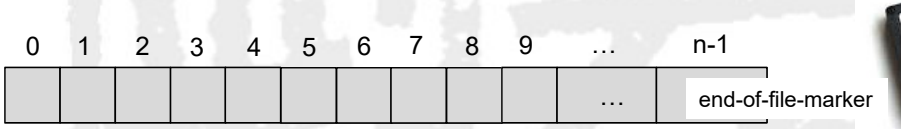
- **I/O** refers to program input and output
  - Can be screen/keyboard or files used to store programs
- A stream is a flow of data
  - Input stream: Data flows into the program
    - If input stream flows from keyboard, the program will accept data from the keyboard → **cin**
    - If input stream flows from a file, the program will accept data from the file
  - Output stream: Data flows out of the program
    - To the screen → **cout**
    - To a file
- Include *<iostream>* for cin/cout, and include *<fstream>* for files

Why Use Files?

- Files allow you to store data permanently!
  - Data in memory will disappear after losing power
- Data output to a file lasts after the program ends
  - Allow data exchange between different programs
- An input file can be used over and over
  - No typing of data again and again for testing
  - Create a data file or read an output file by using the tools that familiar to you
- Files allow you to deal with larger data sets

File Structure

- C++ views each file as a sequence of bytes
- Each file ends either with an end-of-file marker or at a specific byte number recorded in operating
- When a file is opened, an object is created, and a stream is associated with the object
- The streams associated with these objects provide communication channels between a program and a particular file or device



Stream Variables

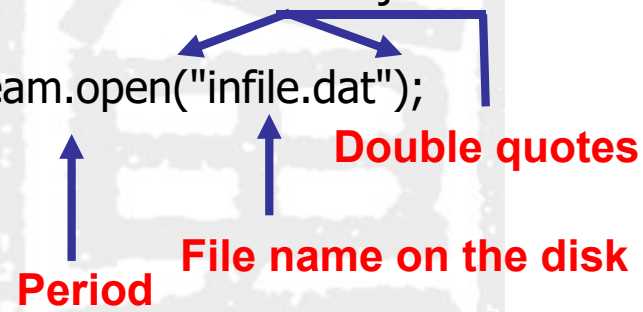
- Like other variables, a stream variable…
  - Must be declared before it can be used
  - Must be initialized before it contains valid data
    - Initializing a stream means connecting it to a file
    - The value of the stream variable can be thought of as the file it is connected to
  - Can have its value changed
    - Changing a stream value means disconnecting from one file and connecting to another
  - Streams use special functions instead of the assignment operator to change values

Declaring Stream Variables

- Input/output file streams are put in another library
  - #include <fstream>
    using namespace std;
- Input-file streams are of type ifstream
  - Ex: declare an input-file stream variable using
    ifstream inStream;
- Output-file streams are of type ofstream
  - Ex: declare an output-file stream variable using
    ofstream outStream;

Connecting To A File

- Once a stream variable is declared, connect it to a file
  - Connecting a stream to a file is opening the file
  - Use the ***open*** function of the stream object



- Once connected to a file, just use the file object as you would use cin/cout

Using the Input/Output Stream

- Using input-stream is similar to using cin with >>
  - Using cin: (from keyboard)

```
int oneNumber, anotherNumber;
cin >> oneNumber >> anotherNumber;
```

  - Using input stream: (from file)

```
inStream.open("infile.dat")
inStream >> oneNumber >> anotherNumber;
```

- An output-stream works similar to the input stream
  - Using cout: (to monitor)

```
int oneNumber, anotherNumber;
cout << oneNumber << anotherNumber;
```

  - Using output stream: (to file)

```
outStream.open("outfile.dat");
outStream << oneNumber << anotherNumber;
```

External File Names

- Is the name for a file in the operating system
  - infile.dat and outfile.dat used in the previous examples
- Is the "real", on-the-disk name for a file
  - Needs to match the naming conventions on your system
- Usually only used in the stream's open statement
- Once open, using the name of the stream object connected to it instead of file name
  - Ex: inStream.open("infile.dat")
    inStream >> oneNumber >> anotherNumber;

real file name

refer to object name

Closing a File

- After using a file, it should be closed
  - This disconnects the stream from the file → released for other file
  - The I/O channels are not unlimited → avoid occupying all resources
  - Reduce the chance of file corruption

- It is important to close an output file if you will read input from the output file later
- The system will automatically close files if you forget as long as your program ends normally

Code: Simple File Input/Output

```
#include <fstream>

int main( )
{
   using namespace std;
   ifstream inStream;
   ofstream outStream;

   inStream.open("infile.dat");
   outStream.open("outfile.dat");

   int first, second, third;
   inStream >> first >> second >> third;
   outStream << "The sum of the first 3\n"
             << "numbers in infile.dat\n"
             << "is " << (first + second + third)
             << endl;

   inStream.close( );
   outStream.close( );

   return 0;
}
```

infile.dat

(Not changed by program.)

```
1
2
3
4
```

outfile.dat

(After program is run.)

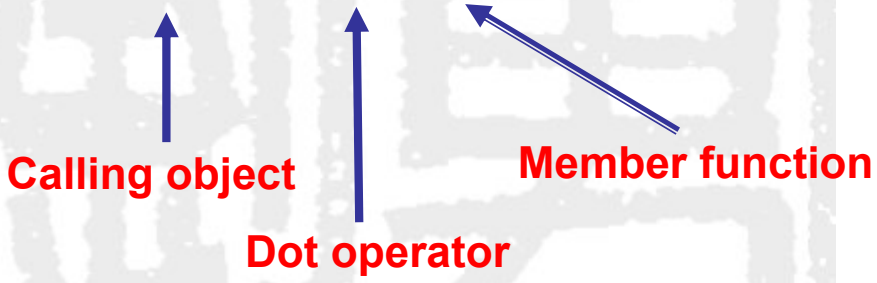
```
The sum of the first 3
numbers in infile.dat
is 6
```

Objects and Member Functions

- An **object** is a variable that has functions and data associated with it
- A **member function** is a function associated with an object
  - *inStream* and *outStream* each have a function named *open* associated with them
  - *inStream* and *outStream* use different versions of a function named *open*
    - One for input files, one for output files
    - They belong to different scopes
- A type whose variables are objects, is a **class**
  - *ifstream* is the type of the *inStream* variable (object)
  - *ifstream* is a class (introduced in Chapter 10)

Calling a Member Function

- Calling a member function requires specifying the object containing the function
- The calling object is separated from the member function by the dot operator
- Example: inStream.open("infile.dat");



Errors on Opening Files

- Opening a file for read could fail for several reasons
  - The file might not exist (wrong file name?)
  - Attempting to read a file without permission
- Opening a file for write could fail for several reasons
  - Attempting to open a file without permission
  - No disk space is available
- May be no error message if file open fails
  - Program execution continues without warning !!
  - Have to check by yourself in the program

Catching Stream Errors

- Member function *fail*, can be used to test the success of a stream operation
  - *fail* returns a boolean type (true or false)
  - TRUE if the stream operation failed
- Immediately following the call to open, check that the operation was successful:

```
inStream.open("stuff.dat");
if( inStream.fail( ) )
 {
        cout << "Input file opening failed.\n";
        exit(1) ;
 }
```

Terminate the program immediately !!

Halting Execution

- When a stream open function fails, it is generally best to stop the program
- The function *exit*, halts a program
  - *exit* returns its argument to the operating system
  - *exit* causes program execution to stop immediately
  - *exit* is NOT a member function
- *Exit* requires including extra library and using directive

```
#include <cstdlib>
using namespace std;
```

Code: File I/O with Checks on Open

```
#include <fstream>
#include <iostream>
#include <cstdlib>

int main( )
{
   using namespace std;
   ifstream inStream;
   ofstream outStream;

   inStream.open("infile.dat");
   if ( inStream.fail( ) )
   {
      cout << "Input file opening failed.\n";
      exit(1);
   }

   outStream.open("outfile.dat");
   if ( outStream.fail( ) )
   {
      cout << "Output file opening failed.\n";
      exit(1);
   }

   int first, second, third;
   inStream >> first >> second >> third;
   outStream << "The sum of the first 3\n"
             << "numbers in infile.dat\n"
             << "is " << (first + second + third)
             << endl;

   inStream.close( );
   outStream.close( );

   return 0;
}
```

Screen Output (If the file infile.dat does not exist)

```
Input file opening failed.
```

Appending Data

- Output examples so far create new files
  - If the output file already exists, its original data is lost
- To append new output to the end an existing file
  - use the open mode ios::app:
    outStream.open("important.txt", ios::app);
  - If the file does not exist, a new file will be created

Mode	Description
ios::app	Append all output to the end of the file.
ios::ate	Open a file for output and move to the end of the file (normally used to append data to a file). Data can be written anywhere in the file.
ios::in	Open a file for input.
ios::out	Open a file for output.
ios::trunc	Discard the file's contents (this also is the default action for ios::out).
ios::binary	Open a file for binary (i.e., nontext) input or output.

Code: Appending to a File

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

int main( )
{
   using namespace std;

   cout << "Opening data.txt for appending.\n";
   ofstream fout;

   fout.open("data.txt", ios::app);

   if (fout.fail( ))
   {
      cout << "Input file opening failed.\n";
      exit(1);
   }

   fout << "5 6 pick up sticks.\n"
        << "7 8 ain't C++ great!\n";

   fout.close( );
   cout << "End of appending to file.\n";

   return 0;
}
```

Sample Dialogue

data.txt
(Before program is run.)

```
1 2 bucket my shoe.
3 4 shut the door.
```

data.txt
(After program is run.)

```
1 2 bucket my shoe.
3 4 shut the door.
5 6 pick up sticks.
7 8 ain't C++ great!
```

Screen Output

```
Opening data.txt for appending.
End of appending to file.
```

File Names as Input

- Program users can enter the name of a file to use for input or for output
- Program must use a variable that can hold multiple characters → string
  - Declaring a variable to hold a string of characters:

    char fileName[16]; ⟶ you can use type string instead
    - Brackets enclose the maximum number of characters + 1
    - The variable fileName contains up to 15 characters
  - Array and string will be introduced in Ch.7 and Ch.8
- How to use filenames as input?

```
cin >> fileName;
inStream.open(fileName);
```

Code: Inputting a File Name

```
#include <fstream>
#include <iostream>
#include <cstdlib>

int main( )
{
    using namespace std;
    char inFileName[16], outFileName[16];
    ifstream inStream;
    ofstream outStream;

    cout << "I will sum three numbers taken from an input\n"
         << "file and write the sum to an output file.\n";
    cout << "Enter the input file name (maximum of 15"
         << " characters):\n";
    cin >> inFileName;
    cout << "Enter the output file name (maximum of 15"
    cout << " characters):\n";
    cin >> outFileName;
    cout << "I will read numbers from the file "
        << inFileName << " and\n"
        << "place the sum in the file "
        << outFileName << endl;
    inStream.open(inFileName);
      if (inStream.fail( ))
      {
         cout << "Input file opening failed.\n";
         exit(1);
      }
    outStream.open(outFileName);
      if (outStream.fail( ))
      {
         cout << "Output file opening failed.\n";
         exit(1);
      }
      int first, second, third;
      inStream >> first >> second >> third;
      outStream << "The sum of the first 3\n"
            << "numbers in " << inFileName << endl
            << "is " << (first + second + third)
            << endl;
      inStream.close( );
      outStream.close( );

      cout << "End of Program.\n";
      return 0;
}
```

Stream Names as Arguments

- Streams can be arguments to a function
  - The function's formal parameter for the stream must be **call-by-reference**
  - You are only allowed to have one "real" channel to a file → cannot be copied as two stream objects
- Example:

```
void makeNeat(ifstream& messyFile, ofstream& neatFile);
```

The End of a File

- Input files used by a program may vary in length
  - Programs may not be able to assume the number of items in the file
- A way to know the end of the file is reached:
  - The boolean expression (inStream >> next)
    - TRUE if a value can be read and stored in next
    - FALSE if there is not a value to be read (the end of the file)
- Example:

```
double next, sum = 0;
int count = 0;
while ( inStream >> next )
{
    sum = sum + next;
    count++;
}
double average = sum / count;
```

To calculate the average of the numbers in a file (file content is unknown)

Detecting the End of a File

- End of a file is indicated by a special character
  - It is often called as **EOF** (End-Of-File)
- Member function *eof* detects the end of a file
  - Member function of every input-file stream
  - *eof* returns a boolean value
    - TRUE when the end of the file has been reached
    - FALSE when there is more data to read
    - After the last character of data is read, eof still returns TRUE until the next character (EOF) is read
  - Normally used to determine when we are NOT at the end of the file
    - Example: if (! inStream.eof( ))

How to Test the End of a File?

- Using boolear expression, ex: (inStream >> next)
- Using eof function, ex:

```
inStream.get(next);
while (! inStream.eof( ) )
{
    cout << next;
    inStream.get(next);
}
```

- Which should be used?
  - In general, use eof when input is treated as text and using a member function get to read input
  - In general, use the extraction operator method when processing numeric data

Overview

6.1 Streams and Basic File I/O

6.2 Tools for Stream I/O

6.3 Character I/O

6.4 C-Style Sequential File Access

Ref: H. M. Deitel and P. J. Deitel, "C How to Program", 5th Ed., Prentice Hall Inc., 2007.

Tools for Stream I/O

- To control the format of the program's output
  - We use member functions or manipulators to determine such details as:
    - Spacing, numeric style, left/right justification, …
- A manipulator is a function called in a non-traditional way
  - Used after the insertion operator (<<) as if the manipulator function call is an output item
  - Defined in the <iomanip> library
- Formatting output to a file uses the similar way as formatting output to the screen (just replace cout)
  - Ex: cout.setf(ios::fixed) → outStream.setf(ios::fixed)
- C-style output commands (printf) is also introduced

C-Style Output --- printf

- printf: precise output formatting
  - Conversion specifications: flags, field widths, precisions, etc.
- Format
  - printf(*format-control-string, other-arguments*);
  - Format control string: describes output format
  - Other-arguments: correspond to each conversion specification in format-control-string
    - Each specification begins with a percent sign(%), ends with conversion specifier
- Comparison to cout:
  - printf("%d\n", int1); ←→ cout << int1 << endl;
  - You have to specify the type and format manually!!

Printing Integers (C style)

- Integer
  - Whole number (no decimal point): 25, 0, -9
  - Positive, negative, or zero
  - Only minus sign prints by default (later we will change this)

Conversion specifier	Description
`d`	Display as a signed decimal integer.
`i`	Display as a signed decimal integer. [*Note:* The `i` and `d` specifiers are different when used with `scanf`.]
`o`	Display as an unsigned octal integer.
`u`	Display as an unsigned decimal integer.
`x` or `X`	Display as an unsigned hexadecimal integer. `X` causes the digits `0-9` and the letters `A-F` to be displayed and `x` causes the digits `0-9` and `a-f` to be displayed.
`h` or `l` (letter `l`)	Place before any integer conversion specifier to indicate that a `short` or `long` integer is displayed, respectively. Letters `h` and `l` are more precisely called **length modifiers**.

Code for Printing Integers (C style)

```
/* Using the integer conversion specifiers */
#include <stdio.h>

int main( void )
{
   printf( "%d\n", 455 );
   printf( "%i\n", 455 );  /* i same as d in printf */
   printf( "%d\n", +455 );
   printf( "%d\n", -455 );
   printf( "%hd\n", 32000 );
   printf( "%ld\n", 2000000000L );  /* L suffix means long int */
   printf( "%o\n", 455 );
   printf( "%u\n", 455 );
   printf( "%u\n", -455 );
   printf( "%x\n", 455 );
   printf( "%X\n", 455 );

   return 0; /* indicates successful termination */
}
```

C include different library for I/O

d and **i** specify signed integers

h specifies a **short** number

l specifies a **long** number

o specifies an octal integer

u specifies an unsigned integer

x and **X** specify hexadecimal integers

output

```
455
455
455
-455
32000
2000000000
707
455
4294966841
1c7
1C7
```


Integral Stream Base in C++

- Change a stream's integer base by inserting manipulators
  - **hex** manipulator
    - Sets the base to hexadecimal (base 16)
  - **oct** manipulator
    - Sets the base to octal (base 8)
  - **dec** manipulator
    - Resets the base to decimal
  - **setbase** parameterized stream manipulator
    - Takes one integer argument: 10, 8 or 16
    - Sets the base to decimal, octal or hexadecimal
    - Requires the inclusion of the <iomanip> header file
  - Stream base values are sticky
    - Remain until explicitly changed to another base value

Code: Change Numerical Base

```
#include <iostream>
using std::cin;
using std::cout;
using std::dec;
using std::endl;
using std::hex;
using std::oct;

#include <iomanip>
using std::setbase;
```

```
int main()
{
   int number;

   cout << "Enter a decimal number: ";
   cin >> number; // input number

   // use hex stream manipulator to show hexadecimal number
   cout << number << " in hexadecimal is: " << hex
      << number << endl;

   // use oct stream manipulator to show octal number
   cout << dec << number << " in octal is: "
      << oct << number << endl;

   // use setbase stream manipulator to show decimal number
   cout << setbase( 10 ) << number << " in decimal is: "
      << number << endl;

   return 0;
} // end main
```

Set base to hexadecimal

Set base to octal

Reset base to decimal

output

```
Enter a decimal number: 20
20 in hexadecimal is: 14
20 in octal is: 24
20 in decimal is: 20
```


Show Integral Stream Base

- Integral base with stream insertion
  - Manipulators dec, hex and oct
- Integral base with stream extraction
  - Integers prefixed with 0 (zero) → octal values
  - Integers prefixed with 0x or 0X → hexadecimal values
  - All other integers → treated as decimal values
- Stream manipulator **showbase**
  - Forces integral values to be outputted with their bases
    - Decimal numbers are output by default
    - Leading 0 for octal numbers
    - Leading 0x or 0X for hexadecimal numbers
  - Reset the showbase setting with **noshowbase**

Code for Showing Number Base

```
// Using stream-manipulator showbase.
#include <iostream>
using namespace std;

int main()
{
   int x = 100;

   // use showbase to show number base
   cout << "Printing integers preceded by their base:" << endl
      << showbase;

   cout << x << endl; // print decimal value
   cout << oct << x << endl; // print octal value
   cout << hex << x << endl; // print hexadecimal value
} // end main
```

output

```
Printing integers preceded by their base:
100
0144
0x64
```


Printing Floating-Point Numbers (C style)

- Floating Point Numbers
  - Have a decimal point (33.5)
  - Exponential notation (computer's version of scientific notation)
    - 150.3 is 1.503 x 10^2 in scientific
    - 150.3 is 1.503E+02 in exponential

Conversion specifier	Description
e or E	Display a floating-point value in exponential notation.
f	Display floating-point values in fixed-point notation.
g or G	Display a floating-point value in either the floating-point form f or the exponential form e (or E), based on the magnitude of the value.
L	Place before any floating-point conversion specifier to indicate that a long double floating-point value is displayed.

Code for Printing FP Numbers (C style)

```
/* Printing floating-point numbers with
   floating-point conversion specifiers */

#include <stdio.h>

int main( void )
{
   printf( "%e\n", 1234567.89 );
   printf( "%e\n", +1234567.89 );
   printf( "%e\n", -1234567.89 );
   printf( "%E\n", 1234567.89 );
   printf( "%f\n", 1234567.89 );
   printf( "%g\n", 1234567.89 );
   printf( "%G\n", 1234567.89 );

   return 0; /* indicates successful termination */

} /* end main */
```

e and **E** specify exponential notation

f specifies fixed-point notation

g and **G** specify either exponential or fixed-point notation depending on the number's size

output

```
1.234568e+006
1.234568e+006
-1.234568e+006
1.234568E+006
1234567.890000
1.23457e+006
1.23457E+006
```

Floating-Point Numbers in C++

- *cout* has member functions to specify the FP format
  - setf(ios::fixed) → specify fixed point notation (ex: 78.5)
  - setf(ios::scientific) → scientific notation (ex: 7.85e01)
  - setf(ios::showpoint) → always show decimal point (75->75.0)
  - precision(2) → two decimal places are shown (ex: 78.50)
- Can also use stream manipulators to set FP format
  - scientific → makes FP numbers display in scientific format
  - fixed → makes FP numbers display with a specific number of digits
  - setprecision(n) → specifies the number of digits to be shown
- Without either scientific or fixed
  - FP number's value determines the output format

Code for Changing FP Format

```
#include <iostream>
using std::cout;
using std::endl;
using std::fixed;
using std::scientific;
int main()
{
   double x = 0.001234567;
   double y = 1.946e9;

   // display x and y in default format
   cout << "Displayed in default format:" << endl
      << x << '\t' << y << endl;

   // display x and y in scientific format
   cout << "\nDisplayed in scientific format:" << endl
      << scientific << x << '\t' << y << endl;

   // display x and y in fixed format
   cout << "\nDisplayed in fixed format:" << endl
      << fixed << x << '\t' << y << endl;
} // end main
```

output

```
Displayed in default format:
0.00123457      1.946e+009

Displayed in scientific format:
1.234567e-003   1.946000e+009

Displayed in fixed format:
0.001235        1946000000.000000
```


Uppercase/Lowercase Control

- Stream manipulator uppercase
  - Causes hexadecimal-integer values to be output with uppercase X and A-F
  - Causes scientific-notation floating-point values to be output with uppercase E
  - These letters output as lowercase by default
  - Reset uppercase setting with nouppercase
- Ex:

```
#include <iostream>
using namespace std;

int main()
{
   cout << "Printing uppercase letters in scientific" << endl
      << "notation exponents and hexadecimal values:" << endl;

   cout << uppercase << 4.345e10 << endl
      << hex << showbase << 123456789  << endl;
} // end main
```

4.345E+010

0X75BCD15

Set Field Widths and Precision (C style)

- Field width: size of field in which data is printed
  - If width larger than data, default right justified
    - Minus sign uses one character position in field
  - If field width too small, increases to fit data
  - Format: insert an integer width between % and specifier
    - Ex: %4d (field width of 4)
- Precision: (Meaning varies depending on data type)
  - Integers: minimum number of digits to print (default: 1)
    - If data too small, prefixed with zeros
  - Floating point:
    - Maximum number of digits to appear after decimal
  - Strings: max number of characters to be written
  - Format: use a dot (.) before precision
    - Ex: %.3f (3 digits after decimal)

Code: Width Setting on Output (C style)

```c
/* Printing integers right-justified */
#include <stdio.h>

int main( void )
{
   printf( "%4d\n", 1 );
   printf( "%4d\n", 12 );
   printf( "%4d\n", 123 );
   printf( "%4d\n", 1234 );
   printf( "%4d\n\n", 12345 ); /* data too large */

   printf( "%4d\n", -1 );
   printf( "%4d\n", -12 );
   printf( "%4d\n", -123 );
   printf( "%4d\n", -1234 ); /* data too large */
   printf( "%4d\n", -12345 ); /* data too large */

   return 0; /* indicates successful termination */

} /* end main */
```

A field width of 4 will make C attempt to print the number in a 4-character space

Note that C considers the minus sign a character

The field width does not work if the provided width is not enough !!

output

```
   1
  12
 123
1234
12345

  -1
 -12
-123
-1234
-12345
```

Code: Precision Setting on Output (C style)

```
/* Using precision while printing numbers */
#include <stdio.h>

int main( void )
{
   int i = 873;              /* initialize int i */
   double f = 123.94536;     /* initialize double f */
   char s[] = "Happy Birthday"; /* initialize char array s */

   printf( "Using precision for integers\n" );
   printf( "\t%.4d\n\t%.9d\n\n", i, i );

   printf( "Using precision for floating-point numbers\n" );
   printf( "\t%.3f\n\t%.3e\n\t%.3g\n\n", f, f, f );

   printf( "Using precision for strings\n" );
   printf( "\t%.11s\n", s );

   return 0;

} /* end main */
```

Precision for integers specifies the minimum number of characters to be printed

Precision for **f** and **e** specifiers controls the number of digits after the decimal point

Precision for the **g** specifier controls the maximum number of significant digits printed

Precision for strings specifies the maximum number of characters to be printed

output

```
Using precision for integers
        0873
        000000873

Using precision for floating-point numbers
        123.945
        1.239e+002
        124

Using precision for strings
        Happy Birth
```


Floating-Point Precision in C++

- Using member function:
  - precision(n): display n digits after the decimal point
  - width(n): set field width as n
- Using stream manipulator:
  - setprecision(n): set precision as n digits
  - setw(n): set field width as n
- For *istream*, width = max no. inputted characters + 1
  - Leave the last space to the end-of-string character (NULL)
- Precision settings are sticky, but width setting are not sticky
  - Remain until explicitly changed

Code: Width Setting on Output

```
#include <iostream>
#include <iomanip>
#include <cmath> // for sqrt
using namespace std;

int main()
{
   double root2 = sqrt( 2.0 );
   int places; // precision, vary from 0-6

   cout << "Square root of 2 with precisions 0-6.\n"
      << "Precision set by member function "
      << "precision:" << endl;

   cout << fixed; // use fixed point format

   // display square root using function precision
   for ( places = 0; places <= 6; places++ )
   {
      cout.precision( places );
      cout << root2 << endl;
   } // end for

   cout << "\nPrecision set by stream manipulator "
      << "setprecision:" << endl;

   // set precision for each digit
   for ( places = 0; places <= 6; places++ )
      cout << setprecision( places ) << root2 << endl;
} // end main
```

output

```
Square root of 2 with precisions 0-6.
Precision set by member function precision:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214

Precision set by stream manipulator setprecision:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
```


Code: Width Setting on Input

```
#include <iostream>
using namespace std;

int main()
{
   int widthValue = 4;
   char sentence[ 10 ];

   cout << "Enter a sentence:" << endl;
   cin.width( 5 ); // read in only 4 characters

   // set field width, then display characters
   while ( cin >> sentence )
   {
      cout.width( widthValue++ );
      cout << sentence << endl;
      cin.width( 5 ); // read in 4 more characters
   } // end while
} // end main
```

output

```
Enter a sentence:
This is a test of the width member function
This
   is
     a
   test
      of
      the
      widt
         h
       memb
          er
         func
(right justified)  tion
```

width=4

width=15

Using Flags in `printf` (C style)

- Flags
  - Supplement formatting capabilities
  - Place flag immediately to the right of percent sign
  - Several flags may be combined

Flag	Description
- (minus sign)	Left justify the output within the specified field.
+ (plus sign)	Display a plus sign preceding positive values and a minus sign preceding negative values.
space	Print a space before a positive value not printed with the + flag.
#	Prefix 0 to the output value when used with the octal conversion specifier o.
	Prefix 0x or 0X to the output value when used with the hexadecimal conversion specifiers x or X.
	Force a decimal point for a floating-point number printed with e, E, f, g or G that does not contain a fractional part. (Normally the decimal point is printed only if a digit follows it.) For g and G specifiers, trailing zeros are not eliminated.
0 (zero)	Pad a field with leading zeros.

Example: Left Justify and Sign (C style)

```
/* Right justifying and left justifying values */
#include <stdio.h>

int main( void )
{
   printf( "%10s%10d%10c%10f\n\n", "hello", 7, 'a', 1.23 );
   printf( "%-10s%-10d%-10c%-10f\n", "hello", 7, 'a', 1.23 );

   return 0; /* indicates successful termination */

} /* end main */
```

- flag left justifies characters in a field

output

```
     hello         7         a  1.230000

hello     7         a         1.230000
```

```
/* Printing numbers with and without the + flag */
#include <stdio.h>

int main( void )
{
   printf( "%d\n%d\n", 786, -786 );
   printf( "%+d\n%+d\n", 786, -786 );

   return 0; /* indicates successful termination */

} /* end main */
```

+ flag forces a plus sign on positive numbers

output

```
786
-786
+786
-786
```

Example: Space Fill or Zero Fill (C style)

```
/* Printing a space before signed values
   not preceded by + or - */
#include <stdio.h>

int main( void )
{
   printf( "% d\n% d\n", 547, -547 );

   return 0; /* indicates successful termination */

} /* end main */
```

Space flag forces a space on positive numbers

output

```
 547
-547
```

```
/* Printing with the 0( zero ) flag fills in leading zeros */
#include <stdio.h>

int main( void )
{
   printf( "%+09d\n", 452 );
   printf( "%09d\n", 452 );

   return 0; /* indicates successful termination */

} /* end main */
```

0 flag fills empty spaces with zeros

output

```
+00000452
000000452
```

Example: Showbase on Output (C style)

```
/* Using the # flag with conversion specifiers
   o, x, X and any floating-point specifier */
#include <stdio.h>

int main( void )
{
   int c = 1427;      /* initialize c */
   double p = 1427.0; /* initialize p */

   printf( "%#o\n", c );
   printf( "%#x\n", c );
   printf( "%#X\n", c );
   printf( "\n%g\n", p );
   printf( "%#g\n", p );

   return 0;

} /* end main */
```

flag prefixes a 0 before octal integers

flag prefixes a 0x before hexadecimal integers

flag forces a decimal point on floating-point numbers with no fractional part

output

```
02623
0x593
0X593

1427
1427.00
```

Trailing Zeros and Decimal Points

```
// Controlling the printing of trailing zeros and
// decimal points in floating-point values.
#include <iostream>
using namespace std;

int main()
{
   // display double values with default stream format
   cout << "Before using showpoint" << endl
      << "9.9900 prints as: " << 9.9900 << endl
      << "9.9000 prints as: " << 9.9000 << endl
      << "9.0000 prints as: " << 9.0000 << endl
      << endl;
   // display double value after showpoint
   cout << showpoint
      << "After using showpoint" << endl
      << "9.9900 prints as: " << 9.9900 << endl
      << "9.9000 prints as: " << 9.9000 << endl
      << "9.0000 prints as: " << 9.0000 << endl;
} // end main
```

- Stream manipulator showpoint
  - Output with decimal point and trailing zeros
  - Ex: 79.0 prints as 79.0000 instead of 79
- Reset showpoint setting with noshowpoint

```
Before using showpoint
9.9900 prints as: 9.99
9.9000 prints as: 9.9
9.0000 prints as: 9

After using showpoint
9.9900 prints as: 9.99000
9.9000 prints as: 9.90000
9.0000 prints as: 9.00000
```


Justification (left, right and internal)

- Justification in a field
  - Manipulator **left**
    - fields are left-justified
    - padding characters to the right
  - Manipulator **right**
    - fields are right-justified
    - padding characters to the left
  - Manipulator **internal**
    - signs or bases on the left
      - showpos forces the plus sign to print
    - magnitudes on the right
    - padding characters in the middle

Code: Different Justification

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   int x = 12345;

   // display x right justified (default)
   cout << "Default is right justified:" << endl
      << setw( 10 ) << x;

   // use left manipulator to display x left justified
   cout << "\n\nUse std::left to left justify x:\n"
      << left << setw( 10 ) << x;

   // use right manipulator to display x right justified
   cout << "\n\nUse std::right to right justify x:\n"
      << right << setw( 10 ) << x << endl;
} // end main
```

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   // display with spacing and plus sign
   cout << internal << showpos
        << setw( 10 ) << 123 << endl;
} // end main
```

```
+       123
```

```
Default is right justified:
     12345

Use std::left to left justify x:
12345

Use std::right to right justify x:
     12345
```


Padding Characters (fill, setfill)

- Padding: fill the empty space in a field
  - Default padding character is space
  - You can specify any character to pad a field
- Member function **fill**(char)
  - Specifies the fill character
  - Returns the prior fill character
- Stream manipulator **setfill**(char)
  - Specifies the fill character
- Padding setting is sticky
  - Have to change back to space by yourself later

Code: Padding Characters

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   int x = 10000;

   // display x
   cout << x
      << " printed using the default\n"
      << " pad character (space):\n";

   // display x with base
   cout << showbase << setw( 10 ) << x << endl;

   // display x with left justification
   cout << left << setw( 10 ) << x << endl;

   // display x as hex with internal justification
   cout << internal << setw( 10 ) << hex << x << endl;
   cout << endl << "Using various padding characters:\n";

   // padded characters (right justification)
   cout << right;
   cout.fill( '*' );
   cout << setw( 10 ) << dec << x << endl;

   // padded characters (left justification)
   cout << left << setw( 10 )
        << setfill( '%' ) << x << endl;

   // padded characters (internal justification)
   cout << internal << setw( 10 )
        << setfill( '^' ) << hex << x
        << endl;
} // end main
```

```
10000 printed using the default
pad character (space):
     10000
10000
0x    2710

Using various padding characters:
*****10000
10000%%%%%
0x^^^^2710
```

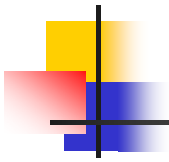

Unsetting Flags

- Any flag that is set, may be unset
- For the flag setting via member functions, use unsetf function to clear the setting (back to default)
  - Example:

    cout.unsetf(ios::showpos);

    causes the program to stop printing plus signs on positive numbers
- The manipulator resetiosflags behaves in the similar way
  - Example:

    resetiosflags(ios::showpos);



Overview

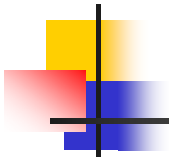
6.1 Streams and Basic File I/O

6.2 Tools for Stream I/O

6.3 Character I/O

6.4 C-Style Sequential File Access

Ref: H. M. Deitel and P. J. Deitel, "C How to Program", 5th Ed., Prentice Hall Inc., 2007.



Low Level Character I/O

- All data is input and output as characters
  - Output of the number 10 is two characters '1' and '0'
  - Input of the number 10 is also done as '1' and '0'
  - Interpretation of 10 as the number 10 or as characters depends on the program
    - cin will automatically convert it into numbers
- Low level C++ functions for character I/O
  - Perform character input and output
  - Do not perform automatic conversions
  - Allow you to do input and output in anyway you can devise

Member Function get

- Function *get*
  - Reads one character from an input stream
  - Stores the character read in a variable of type char
  - Does not use the extraction operator (>>)
  - Does not skip blanks
- These lines use *get* to read a character and store it in the variable *nextSymbol*

```
char nextSymbol;
cin.get(nextSymbol);   or
inStream.get(nextSymbol);
```

  - Any character will be read with these statements
    - Blank spaces too!
    - '\n' too! (The newline character)

Example: cin vs cin.get

- Given this code:

```
char c1, c2, c3;
cin.get(c1);
cin.get(c2);
cin.get(c3);
```

  and this input:

```
AB
CD
```

  → c1 = 'A' c2 = 'B' c3 = '\n'
  - cin >> c1 >> c2 >> c3; would place 'C' in c3 (the ">>" operator skips the newline character)
- '\n' vs "\n"
  - '\n' : a character value → stored in a variable of type char
  - "\n" : a string containing only one character → cannot be stored in a variable of type char

Member Function put

- Function *put*
  - Requires one argument of type char
  - Sends its argument to the output stream
  - Similar effects as using cout + insertion operator
- Examples:

```
cout.put(nextSymbol);
cout.put('a');

ofstream outStream;
outStream.open("outfile.dat");
outStream.put('Z');
```

Member Function putback

- The *putback* member function places a character back to the input stream (for next read)
  - Useful when input continues until a specific character is read, but you do not want to process the character
  - Places its argument of type char in the input stream
    - Does not have to be a character read from the stream
- Ex: read input stream until space, but put the space back

```
fin.get(next);
while (next != ' ')
{
        fout.put(next);
        fin.get(next);
}
fin.putback(next);
```

Example: Checking Input

- Incorrect input can produce worthless output
- Use input functions that allow the user to re-enter input until it is correct, such as
  - Echoing the input and asking the user if it is correct
  - If the input is not correct, allow user to enter the data again
- The *getInt* function seen in Display 6.7 obtains an integer value from the user
  - Prompts the user, reads the input, and displays the input
  - After displaying the input, asks the user to confirm the number and reads the user's response
  - The process is repeated until the user indicates with a 'Y' or 'y' that the number entered is correct

Code: Checking Input

```
#include <iostream>
using namespace std;

void newLine( );
//Discards all the input remaining on the current
//input line and the '\n' at the end of the line.

void getInt(int& number);
//Postcondition: The variable number has been
//given a value that the user approves of.

int main( )
{
    int n;

    getInt(n);
    cout << "Final value read in = " << n << endl
         << "End of demonstration.\n";
    return 0;
}
```

Sample Dialogue

```
Enter input number: 57
You entered 57. Is that correct? (yes/no): No
Enter input number: 75
You entered 75. Is that correct? (yes/no): yes
Final value read in = 75
End of demonstration.
```

```
//Uses iostream:
void newLine( )
{
    char symbol;
    do
    {
        cin.get(symbol);
    } while (symbol != '\n');
}

//Uses iostream:
void getInt(int& number)
{
    char ans;
    do
    {
        cout << "Enter input number: ";
        cin >> number;
        cout << "You entered " << number
             << " Is that correct? (yes/no): ";
        cin >> ans;
        newLine( );
    } while ((ans != 'Y') && (ans != 'y'));
}
```

Checking Input: Yes or No?

- *getInt* continues to ask for a number until the user responds 'Y' or 'y' using the do-while loop

```
do
{
   // the loop body
} while  ((ans !='Y') &&(ans != 'y') )
```

- Why not use ((ans =='N') | | (ans == 'n'))?
  - User must enter a correct response to continue a loop tested with ((ans =='N') | | (ans == 'n'))
    - What if they mis-typed "Bo" instead of "No"?
  - User must enter a correct response to end the loop tested with ((ans !='Y') &&(ans != 'y'))
    - Keep accepting input if they mis-typed the answers

Checking Input : newLine

- The *newLine* function seen in Display 6.7 is called by the *getInt* function
  - *newLine* reads all the characters remaining in the input line and discards them
  - Discard what follows the first character of the user's response to "Is that correct? (yes/no)"
    - The newline character is discarded as well
- Be sure to deal with the '\n' that ends each input line if using cin >> and cin.get
  - "cin >>" reads up to '\n' but leaves it in the input stream
  - cin.get will read the '\n' → the newline function in Display 6.7 can be used to clear the '\n'

Printing Strings and Characters (C style)

- The conversion specifiers for string and char:
  - c
    - Prints char argument
    - Cannot be used to print the first character of a string
  - s
    - Requires a pointer to char as an argument
    - Prints characters until NULL ('\0') encountered
    - Cannot print a char argument
  - Remember
    - Single quotes for character constants ('z')
    - Double quotes for strings "z" (which actually contains two characters, 'z' and '\0')

Code Example for c & s (C style)

```
#include <stdio.h>

int main( void )
{
   char character = 'A'; /* initialize char */
   char string[] = "This is a string"; /* initialize char array */
   const char *stringPtr = "This is also a string"; /* char pointer */

   printf( "%c\n", character );
   printf( "%s\n", "This is a string" );
   printf( "%s\n", string );
   printf( "%s\n", stringPtr );

   return 0; /* indicates successful termination */
} /* end main */
```

c specifies a character will be printed

s specifies a string will be printed

```
A
This is a string
This is a string
This is also a string
```

Formatting Input with scanf (C style)

- scanf
  - Input can be formatted much like output can
  - scanf conversion specifiers are slightly different from those used with printf
- Ex:
  - scanf("%d",&int2); ←→ cin >> int2;
  - You have to specify the input format manually!!
  - Provide the pointer for the storage variable instead of the variable itself

Conversion Specifiers for scanf (1/2)

Conversion specifier	Description
Integers	
`d`	Read an optionally signed decimal integer. The corresponding argument is a pointer to an `int`.
`i`	Read an optionally signed decimal, octal or hexadecimal integer. The corresponding argument is a pointer to an `int`.
`o`	Read an octal integer. The corresponding argument is a pointer to an unsigned `int`.
`u`	Read an unsigned decimal integer. The corresponding argument is a pointer to an unsigned `int`.
`x` or `X`	Read a hexadecimal integer. The corresponding argument is a pointer to an unsigned `int`.
`h` or `l`	Place before any of the integer conversion specifiers to indicate that a `short` or `long` integer is to be input.

Conversion Specifiers for scanf (2/2)

Conversion specifier	Description
Floating-point numbers	
`e`, `E`, `f`, `g` or `G`	Read a floating-point value. The corresponding argument is a pointer to a floating-point variable.
`l` or `L`	Place before any of the floating-point conversion specifiers to indicate that a `double` or `long double` value is to be input. The corresponding argument is a pointer to a `double` or `long double` variable.
Characters and strings	
`c`	Read a character. The corresponding argument is a pointer to a `char`; no null (`'\0'`) is added.
`s`	Read a string. The corresponding argument is a pointer to an array of type `char` that is large enough to hold the string and a terminating null (`'\0'`) character—which is automatically added.

Example: Use scanf for Input Data (C style)

```
#include <stdio.h>

int main( void )
{
   int a;
   int b;
   int;
   int cd;
   int e;
   int f;
   int g;

   printf( "Enter seven integers: " );
   scanf( "%d%i%i%i%o%u%x", &a, &b, &c, &d, &e, &f, &g );

   printf( "The input displayed as decimal integers is:\n" );
   printf( "%d %d %d %d %d %d %d\n", a, b, c, d, e, f, g );

   return 0;

} /* end main */
```

`d` specifies a decimal integer will be input

`i` specifies an integer will be input

`o` specifies an octal integer will be input

`u` specifies an unsigned decimal integer will be input

`x` specifies a hexadecimal integer will be input

output

```
Enter seven integers: -70 -70 070 0x70 70 70 70
The input displayed as decimal integers is:
-70 -70 56 112 56 70 112
```

Character Functions

- Several predefined functions exist to facilitate working with characters
- The cctype library is required
  - #include <cctype>
    using namespace std;

Some Predefined Character Functions in cctype (*part 1 of 2*)

Function	Description	Example
toupper(*Char_Exp*)	Returns the upper-case version of *Char_Exp*.	*char* c = toupper('a'); cout << c; **Outputs:** A
tolower(*Char_Exp*)	Returns the lower-case version of *Char_Exp*.	*char* c = tolower('A'); cout << c; **Outputs:** a

The toupper Function

- *toupper* returns the argument's upper case character
  - toupper('a') returns 'A'
  - toupper('A') return 'A'
- *toupper* and *tolower* actually return the integer representing the character
  - cout << toupper('a'); //prints the integer for 'A'
  - char c = toupper('a'); //places the integer for 'A' in c
    cout << c; //prints 'A'
  - cout << static_cast<char>(toupper('a')); //works too

The isspace Function

- *isspace* returns true if the argument is whitespace
  - Whitespace is spaces, tabs, and newlines
  - Example:

```
if (isspace(next) )
    cout << '-';
else
    cout << next;
```

  - Prints a '-' if next contains a space, tab, or newline character
  - More IS functions here →

Function	Description	Example

`isupper(Char_Exp)`	Returns *true* provided *Char_Exp* is an uppercase letter; otherwise, returns *false*.	`if (isupper(c))`
		`  cout << c << " is uppercase.";`
		`else`
		`  cout << c`
		`  << " is not uppercase.";`

`islower(Char_Exp)`	Returns *true* provided *Char_Exp* is a lowercase letter; otherwise, returns *false*.	`char c = 'a';`
		`if (islower(c))`
		`  cout << c << " is lowercase.";`
		Outputs: a is lowercase.

`isalpha(Char_Exp)`	Returns *true* provided *Char_Exp* is a letter of the alphabet; otherwise, returns *false*.	`char c = '$';`
		`if (isalpha(c))`
		`  cout << c << " is a letter.";`
		`else`
		`  cout << c`
		`  << " is not a letter.";`
		Outputs: $ is not a letter.

`isdigit(Char_Exp)`	Returns *true* provided *Char_Exp* is one of the digits '0' through '9'; otherwise, returns *false*.	`if (isdigit('3'))`
		`  cout << "It's a digit.";`
		`else`
		`  cout << "It's not a digit.";`
		Outputs: It's a digit.

`isspace(Char_Exp)`	Returns *true* provided *Char_Exp* is a whitespace character, such as the blank or newline symbol; otherwise, returns *false*.	`//Skips over one "word" and`
		`//sets c equal to the first`
		`//whitespace character after`
		`//the "word":`
		`do`
		`{`
		`  cin.get(c);`
		`} while (! isspace(c));`

Overview

6.1 Streams and Basic File I/O

6.2 Tools for Stream I/O

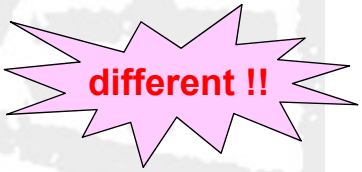
6.3 Character I/O

6.4 C-Style Sequential File Access

Ref: H. M. Deitel and P. J. Deitel, "C How to Program", 5th Ed., Prentice Hall Inc., 2007.

Create a Sequential-Access File (C style)

- C style has no extra library for files
  - #include <stdio.h>
- Creating a File
  - **FILE** **cfPtr*
    - Creates a FILE pointer called *cfPtr*
  - cfPtr = fopen("clients.dat", "w")
    - Function fopen returns a FILE pointer to file specified
    - Takes two arguments – file to open and file open mode
    - If open fails, **NULL** returned
- Read/Write use the same type of FILE pointers



Open Mode for Files in C

Mode	Description
r	Open an existing file for reading.
w	Create a file for writing. If the file already exists, discard the current contents.
a	Append; open or create a file for writing at the end of the file.
r+	Open an existing file for update (reading and writing).
w+	Create a file for update. If the file already exists, discard the current contents.
a+	Append: open or create a file for update; writing is done at the end of the file.
rb	Open an existing file for reading in binary mode.
wb	Create a file for writing in binary mode. If the file already exists, discard the current contents.
ab	Append; open or create a file for writing at the end of the file in binary mode.
rb+	Open an existing file for update (reading and writing) in binary mode.
wb+	Create a file for update in binary mode. If the file already exists, discard the current contents.
ab+	Append: open or create a file for update in binary mode; writing is done at the end of the file.

Read/Write Functions in a File (C style)

- fgetc
  - Reads one character from a file
  - Takes a FILE pointer as an argument
  - fgetc(stdin) equivalent to getchar()
- fputc
  - Writes one character to a file
  - Requires a FILE pointer and a character to write
  - fputc('a', stdout) equivalent to putchar('a')
- fgets
  - Reads a line from a file
- fputs
  - Writes a line to a file
- fscanf / fprintf
  - File processing equivalents of scanf and printf

Other Functions in a File (C style)

- fprintf(*FILE pointer*, format control sequence)
  - Used to print to a file
  - Like printf, except first argument is a FILE pointer (pointer to the file you want to print in)
- feof(*FILE pointer*)
  - Returns true if end-of-file indicator (no more data to process) is set for the specified file
- fclose(*FILE pointer*)
  - Closes specified file
  - Performed automatically when program ends
  - Good practice to close files explicitly
- Each file must have a unique name and should have its own pointer

Reading Data from a File (C style)

- Create a FILE pointer, link it to the file to read
  cfPtr = fopen("clients.dat", "r");
- Use fscanf to read from the file
  - Like scanf, except first argument is a FILE pointer
  fscanf(cfPtr, "%d%s%f", &accounnt, name, &balance);
- Data read from beginning to end
- File position pointer
  - Indicates number of next byte to be read / written
  - Not really a pointer, but an integer value (byte location)
  - Also called byte offset
- rewind(cfPtr)
  - Reset file position pointer to beginning of file (byte 0)

Example: Using FILE in C Style (1/4)

```
1  /* Credit inquiry program */
2   #include <stdio.h>
3
4   /* function main begins program execution */
5   int main( void )
6   {
7         int request;      /* request number */
8         int account;      /* account number */
9       double balance;/* account balance */
10      char name[ 30 ];              /* account name */
11      FILE *cfPtr;      /* clients.dat file pointer */
12
13       /* fopen opens the file; exits program if file cannot be opened */
14      if( ( cfPtr = fopen( "clients.dat", "r" ) ) == NULL ) {
15      printf( "File could not be opened\n" );
16      } /* end if */
17      else {
18
19           /* display request options */
20          printf( "Enter request\n"
21                  " 1 – List accounts with zero balances\n"
22                  " 2 – List accounts with credit balances\n"
23                  " 3 – List accounts with debit balances\n"
24                  " 4 – End of run\n?" );
25          scanf( "%d", &request );
```

fopen function opens a file; **r** argument means the file is opened for reading

Example: Using FILE in C Style (2/4)

```
28          /* process user's request */
29          while ( request != 4 ) {
30
31             /* read account, name and balance from file */
32             fscanf( cfPtr, "%d%s%1f", &account, name, &balance );
33
34             switch ( request ) {
35
36                case 1:
37                   printf( "\nAccounts with zero balances:\n" );
38
39                   /* read file contents (until eof) */
40                   while ( !feof( cfPtr ) ) {
41
42                      if( balance == 0 ) {
43                         printf( "%-10d%-13s%7.2f\n",
44                            account, name, balance );
45                      } /* end if */
46
47                      /* read account, name and balance from file */
48                      fscanf( cfPtr, "%d%s%1f",
49                         &account, name, &balance );
50                   } /* end while */
51
52                   break;
```

`fscanf` function reads a string from a file

Example: Using FILE in C Style (3/4)

```
54                case 2:
55                   printf( "\nAccounts with credit balances:\n" );
56
57                   /* read file contents (until eof) */
58                   while ( !feof( cfPtr ) ) {
59
60                      if ( balance < 0 ) {
61                         printf( "%-10d%-13s%7.2f\n",
62                            account, name, balance );
63                      } /* end if */
64
65                      /* read account, name and balance from file */
66                      fscanf( cfPtr, "%d%s%1f",
67                         &account, name, &balance );
68                   } /* end while */
69
70                   break;
71
72                case 3:
73                   printf( "\nAccounts with debit balances:\n" );
74
75                   /* read file contents (until eof) */
76                   while ( !feof( cfPtr ) ) {
77
78                      if( balance > 0 ) {
79                         printf( "%-10d%-13s%7.2f\n",
80                            account, name, balance );
81                      } /* end if */
```

Example: Using FILE in C Style (4/4)

```
83                          /* read account, name and balance from file */
84                          fscanf( cfPtr, "%d%s%1f",
85                              &account, name, &balance );
86                      } /* end while */
87
88                      break;
89
90                  } /* end switch */
91
92                  rewind( cfPtr ); /* return cfPtr to beginning of file */
93
94                  printf( "\n? " );
95                  scanf( "%d", &request );
96              } /* end while */
97
98              printf( "End of run.\n" );
99              fclose( cfPtr ); /* fclose closes the file */
100         } /* end else */
101
102         return 0; /* indicates successful termination */
103
104     } /* end main */
```

rewind function moves the file pointer back to the beginning of the file

Example: Program Results

```
Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run
? 1

Accounts with zero balances:
300       White           0.00

? 2

Accounts with credit balances:
400       Stone         -42.16

? 3

Accounts with debit balances:
100       Jones          24.98
200       Doe           345.67
500       Rich          224.62

? 4
End of run.
```