

## Justification

- Justification refers to the alignment of data within a horizontal field.
- The default justification in output fields is to the right, with padding occurring first (on the left).
- To reverse the default justification to the left:

```
cout << fixed << showpoint;
string empName = "Flintstone, Fred";
double Wage = 8.43;
double Hours = 37.5;
cout << left; //turn on left justification
cout << setw(20) << empName;
cout << right; //turn on right justification
cout << setw(10) << setprecision(2) << Wage * Hours << endl;
```

This will produce the output:

```
012345678901234567890123456789
Flintstone, Fred          316.13
```

## Padding Output

- Padding refers to the character used to fill in the unused space in an output field.
- By default the pad character for justified output is the space (blank) character.
- This can be changed by using the `setfill()` manipulator:

```
int ID = 413225;
cout << "0123456789" << endl;
cout << setw(10) << ID << endl;
cout << setfill('0'); //pad with zeroes
cout << setw(10) << ID << endl;
cout << setfill(' '); //reset padding to spaces
```

This will produce the output:

```
0123456789
      413225
0000413225
```

```
#include <fstream>
#include <iomanip>
#include <string>
using namespace std;

int main() {

    ofstream outFile("AreaData.out");
    outFile << fixed << showpoint;

    const double PI = 3.141592654;
    string figName = "Ellipse";
    int majorAxis = 10,
        minorAxis = 2;
    double Area = PI * majorAxis * minorAxis;

    outFile << setw(20) << "Area" << endl;
    outFile << left << setw(10) << figName;
    outFile << right << setw(10) << setprecision(4)
        << Area << endl;

    outFile.close();
    return 0;
}
```

Area	
Ellipse	62.8319

When you attempt to read a value from an input stream, the extract operator or other input function takes into account the type of the variable into which the value is to be stored. If there is a default conversion between the type of the data in the stream and the type of the target variable, then that is applied and all is well.

What happens if the next data in the input stream is not compatible with the target variable?

In that case, the input operation fails.

The effect on the target variable is compiler-dependent. With Visual C++, the target variable is generally not modified (`string` variables are an exception).

The stream variable sets an internal flag indicating it is in a "fail state". Subsequent attempts to read from the stream will automatically fail (see `clear()` later in these notes).

In consequence, it is vital that C++ programmers design input code to reflect the formatting of the input data. Programming to handle general, unstructured input data is extremely difficult.

Consider the following input code fragments and associated input streams:

```
int    iA, iB;
double dX, dY;
char   cC;
string sS;

// input code           stream data (space separated)
In >> iA >> iB;        // 17   x   42

In >> dX >> dY;        // 73   .2

In >> iA >> cC >> iB;   // 73   .2

In >> cC >> dX >> sS;   // 42   23bravoxray

In >> iA >> cC >> sS;   // 42   23bravoxray
```

When you attempt to extract a value from an input stream, the stream variable returns an indication of success (true) or failure (false). You can use that to check whether the input operation has failed for some reason.

A `while` loop is used to extract data from the input stream until it fails.

Note well: the design here places a test to determine whether the read attempt succeeded between each attempt to read data and each attempt to process data. Any other approach would be logically incorrect.

## Design Logic

Try to read data. (Often called the priming read.)

While the last attempt to read data succeeded do

    Process the last data that was read.

    Try to read data.

Endwhile

This is one of the most common patterns in programming.

# Failure-Controlled Input Example

Formatted I/O 7

```
. . .
const int MINPERHOUR = 60;
int Time;          // time value in total minutes
int Hours,        // HH field of Time
    Minutes;     // MM field of Time

int numTimes = 0; // number of Time values read
int totalTime = 0; // sum of all Time values

In >> Time;
while ( In ) {
    numTimes++;
    totalTime = totalTime + Time;
    Hours = Time / MINPERHOUR;
    Minutes = Time % MINPERHOUR;
    Out << setw(5) << numTimes << " | ";
    Out << setw(5) << Hours << " : ";
    Out << setw(2) << setfill('0')
        << Minutes << setfill(' ') << endl;

    In >> Time;
}

Out << "Total minutes: " << setw(5) << totalTime
    << endl;
. . .
```

Input:

217
49
110
302
91
109
198

Output:

1	3:37
2	0:49
3	1:50
4	5:02
5	1:31
6	1:49
7	3:18
Total minutes: 1076	

# Failure-Controlled Input Example

The program given on the previous slide will terminate gracefully if the input file contains an error that causes the input operation to fail:

Input:

217
49
110
xxx
91
109
198

Output:

1	3:37
2	0:49
3	1:50
Total minutes: 376	

Trace the execution...

The input-failure logic used in the sample code has more than one virtue:

- it will terminate automatically at the end of a correctly-formatted input file.
- it will terminate automatically if an input failure occurs while reading an incorrectly-formatted input file, acting as if that point were in fact the end of the input file.

Of course, it would be nice if an error message were also generated above...

# Incorrect Design

```
...
// NO priming read
while ( In ) {
    In >> Time;        // Read at beginning of loop

    numTimes++;       // . . . then process data.

    totalTime = totalTime + Time;
    Hours = Time / MINPERHOUR;
    Minutes = Time % MINPERHOUR;

    Out << setw(5) << numTimes << " | ";
    Out << setw(5) << Hours << " : ";
    Out << setw(2) << setfill('0')
        << Minutes << setfill(' ') << endl;
}
...
```

Input:

217
49
110
302
91
109
198

Output:

1	3:37
2	0:49
3	1:50
4	5:02
5	1:31
6	1:49
7	3:18
8	3:18
Total minutes: 1274	

This is a classic error!

Note that the last time value is output twice. That's typical of the results when this design error is made... watch for it.

## String Input: `getline( )`

Formatted I/O 10

The `getline( )` standard library function provides a simple way to read character input into a string variable, controlling the “stop” character.

Suppose we have the following input file:

Fred Flintstone	Laborer	13301
Barney Rubble	Laborer	43583

There is a single tab after the employee name, another single tab after the job title, and a newline after the ID number.

Assuming `iFile` is connected to the input file above, the statement

```
getline(iFile, String1);
```

would result in `String1` having the value:

```
"Fred Flintstone    Laborer          13301"
```

As used on the previous slide, `getline( )` takes two parameters. The first specifies an input stream and the second a string variable.

Called in this manner, `getline( )` reads from the current position in the input stream until a newline character is found.

Leading whitespace is included in the target string.

The newline character is removed from the input stream, but not included in the target string.

It is also possible to call `getline( )` with three parameters. The first two are as described above. The third parameter specifies the “stop” character; i.e., the character at which `getline( )` will stop reading from the input stream. If found, the stop character will be removed from the input stream.

By selecting an appropriate stop character, the `getline( )` function can be used to read text that is formatted using known delimiters. The example program on the following slides illustrates how this can be done with the input file specified on the preceding slide.

# String Input Example

Formatted I/O 12

```
#include <fstream>           // file streams
#include <iostream>          // standard streams
#include <string>             // string variable support
#include <climits>           // using standard library
using namespace std;

int main() {

    string EmployeeName, JobTitle; // strings for name and title
    int EmployeeID;                // int for id number

    ifstream iFile("Employees.data");

    // Priming read:
    getline(iFile, EmployeeName, '\t'); // read to first tab
    getline(iFile, JobTitle, '\t');     // read to next tab
    iFile >> EmployeeID;                // extract id number
    iFile.ignore(INT_MAX, '\n');        // skip to start of next line
    . . .
}
```

# String Input Example

## Formatted I/O 13

```
. . .
while (iFile) {                                // read to input failure

    cout << "Next employee: " << endl;         // print record header
    cout << EmployeeName << endl              // name on one line
        << JobTitle                            // title and id number
        << EmployeeID << endl << endl;       //   on another line

    getline(iFile, EmployeeName, '\t');        // repeat priming read
    getline(iFile, JobTitle, '\t');           //   logic
    iFile >> EmployeeID;
    iFile.ignore(INT_MAX, '\n');
}

iFile.close();                                // close input file
return 0;
}
```

There are a few useful manipulators that have any effect on input streams:

```
#include <iostream>
#include <iomanip>
using namespace std;
. . .
In >> ws;                // read/discard all leading whitespace
                        //   from current stream position

int hValue;
In >> hex >> hValue;     // reads various input formats as
                        //   being in hexadecimal (base 16)
                        // similarly, oct and dec
```

## fail( ) Member Function

Formatted I/O 15

`fail( )` provides a way to check the status of the last operation on the input stream.

`fail( )` returns true if the last operation failed and returns false if the operation was successful.

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

int main( ) {
    ifstream inStream;
    inStream.open("infile.dat");

    if ( inStream.fail() ) {
        cout << "File not found. Please try again." ;
        return 1;
    }
    // . . . omitted statements doing something useful . . .
    inStream.close();
}
```

## `clear( )` Member Function

Formatted I/O 16

`clear( )` provides a way to reset the status flags of an input stream, after an input failure has occurred.

**Note: closing and re-opening the stream does NOT clear its status flags.**

Consider designing a program to read an input file containing statistics for a baseball player, as shown below, and to produce a summary:

Hits	At-Bats
3	4
2	3
1	3
2	4
Hammerin'	Hokie

Unless we know exactly how many lines of batting data are going to be given, we must use an input-failure loop to read the batting data. But then the input stream will be in a fail state, and we still need to read the player's name.

We can use `clear( )` to recover from the input failure and continue reading input.

# Using clear()

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
#include <climits>
using namespace std;

int main() {

    ifstream In("Hitting.data");

    string playerName;           // player's name
    int Hits, atBats;           // # of hits and at-bats in current game
    int numGames = 0;          // # of games reported
    int totalHits = 0,         // total # of hits in all games
        totalAtBats = 0;      // total # of at-bats in all games

    In.ignore(INT_MAX, '\n');   // skip over header line

    In >> Hits >> atBats;      // try to read 1st game data
    while ( In ) {
        totalHits = totalHits + Hits;           // update running totals
        totalAtBats = totalAtBats + atBats;
        numGames++;                             // count this game
        In >> Hits >> atBats;                   // try to read next game data
    }
    . . .
}
```

## Using `clear()`

Formatted I/O 18

```
. . .  
// Recover from the read failure at the end of the  
// batting data:  
In.clear();  
// Read the player's name:  
getline(In, playerName);  
  
// Calculate the batting average:  
double battingAverage = double(totalHits) / totalAtBats;  
  
// Write the results:  
cout << fixed << showpoint;  
cout << playerName << " is batting "  
    << setprecision(3) << battingAverage  
    << " in " << numGames << " games."  
    << endl;  
  
In.close();  
return 0;  
}
```

Every file ends with a special character, called the end-of-file mark.

`eof ( )` is a boolean function that returns true if the last input operation attempted to read the end-of-file mark, and returns false otherwise.

The program on slide 4.24 could be modified as follows to use `eof ( )` to generate an error message if an input failure occurred in the loop:

```
. . .
Out << "Total minutes: " << setw(5) << totalTime
    << endl;

if ( !In.eof() ) {
    Out << endl
        << "An error occurred while reading the file." << endl
        << "Please check the input file." << endl;
}
. . .
```

In general, reading until input failure is safer than reading until the end-of-file mark is reached. DO NOT use `eof ( )` as a substitute for the input-failure logic covered earlier.

## get ( ) Member Function

Formatted I/O 20

The input stream object `cin` has a member function named `get ( )` which returns the next single character in the stream, whether it is whitespace or not.

To call a member function of an object, state the name of the object, followed by a period, followed by the function call:

```
cin.get(someChar);    // where someChar is a char variable
```

This call to the `get ( )` function will remove the next character from the stream `cin` and place it in the variable `someChar`.

So to read all three characters from a stream containing "A M", we could have:

```
cin.get(ch1);        // read 'A'  
cin.get(someChar);  // read the space  
cin.get(ch2);        // read 'M'
```

The `istream` class provides many additional member functions. Here are two that are often useful:

`peek()` provides a way to examine the next character in the input stream, without removing it from the stream.

```
. . .  
char nextCharacter;  
nextCharacter = In.peek();  
. . .
```

`putback()` provides a way to return the last character read to the input stream.

```
. . .  
const char PUTMEBACK = '?';  
char nextCharacter;  
In.get(nextCharacter);  
if ( nextCharacter == PUTMEBACK ) {  
    In.putback(nextCharacter);  
}  
. . .
```