

# **STREAMS AND FILES**

## **OVERVIEW**

# OVERVIEW

- **Many programs are "data processing" applications**
  - Read the input data
  - Perform sequence of operations on this data
  - Write the output data
- **How we read and write this data is a key part of program**
  - Currently we are using cin and cout for input / output
  - Input stream "cin" is used to read user input
  - Output stream "cout" is used to print program output
  - This is only effective for small quantities of data

# OVERVIEW

- **Files are very useful for data processing applications**
  - Files provide long term storage of valuable information
  - Files can contain large quantities of data
  - Files can be viewed and modified by text editors
  - Files can be read and written by programs
- **In this section, we will show how**
  - Files can be used for program input and output
  - Input stream "ifstream" is used to read user input
  - Output stream "ofstream" is used to print program output

# OVERVIEW

- **Lesson objectives:**

- Learn more about input and output streams
- Learn how open and close ASCII files
- Learn how to read and write ASCII files
- Learn about input / output error checking
- Study programs for numerical data input / output
- Study programs for mixed data input / output

# **STREAMS AND FILES**

## **PART 1**

### **STANDARD I/O STREAMS**

# STANDARD I/O STREAMS

- The term “stream” in computer science has a technical definition of “a sequence of data elements made available over time”
  - We can visualize the data elements as moving along a conveyor belt and being processed one at a time
- The “cin” command in C++ is an **input stream**, where characters that are typed by the user are processed one at a time to read variables of different types
- The “cout” command in C++ is an **output stream**, where variable values are converted into ascii characters and then displayed one at a time on the monitor

# STANDARD I/O STREAMS

- **The "cin" command is used to read input from keyboard**
  - Cin skips spaces and newlines before reading value
  - Cin then does ascii to binary conversion to read variable
  - Cin stops reading at space or non-matching character
- **Examples**
  - "cin >> char\_variable" – reads single character
  - "cin >> int\_variable" – reads sequence of digits
  - "cin >> float\_variable" – reads digits and decimal point
  - "cin >> string\_variable" – reads sequence of characters

# STANDARD I/O STREAMS

- **The "cout" command is used to print output on screen**
  - Cout then does binary to ascii conversion on variable
  - Cout prints output characters to screen
  - Cout does not print any spaces before or after variable
- **Examples**
  - "cout << char\_variable" – prints single character
  - "cout << int\_variable" – prints sequence of digits
  - "cout << float\_variable" – prints digits and decimal point
  - "cout << string\_variable" – prints sequence of characters



# STANDARD I/O STREAMS

- **The input command (`cin >> variable`) will return a value**
  - TRUE if the variable is read successfully
  - FALSE if the variable is NOT read successfully
- **(`cin >> variable`) is NOT successful when**
  - End of file has been reached
    - Example: if user enters control-d
  - Unexpected character encountered when reading variable
    - Example: if user types in "hello" instead of an integer

# STANDARD I/O STREAMS

- **Example program for reading sequence of integer values**

```
int num = 0;  
int total = 0;  
while ((cin >> num) && (num >= 0))  
    total += num;  
cout << "total=" << total << endl;
```



Loop will read  
integers until  
eof is reached  
or a negative  
value is read

# SINGLE CHARACTER I/O

- **We can use single character I/O for more control**
  - `cout.put(ch)` will write one character `ch` onto output stream
  - `cin.get(ch)` reads next character from input stream into `ch`
  - `cin.unget()` will undo the last get from the input stream
  - `cin.peek()` will look ahead one character without reading
- **We can do I/O error checking using return values**
  - `cin.eof()` returns `TRUE` if end of file char has been read and `FALSE` otherwise
  - `cin.get(ch)` returns `TRUE` if end of file char has NOT been read and `FALSE` otherwise

# SINGLE CHARACTER I/O

- Character-by-character copy example

```
char Ch;  
cin.get(Ch);  
while ( !cin.eof() )  
{  
    cout.put(Ch);  
    cin.get(Ch);  
}
```

← Will be true after  
an unsuccessful  
read attempt

# SINGLE CHARACTER I/O

- Shorter character-by-character copy example

```
char Ch;  
while (cin.get(Ch))  
    cout.put(Ch);
```

Get returns true if  
read is successful  
and false if end of  
file is reached



# ASCII TO INTEGER CONVERSION

- **In most applications, we can simply use “cin >> num” to read an integer value into a variable**
  - What if we want to do special error checking to make sure the user enters data correctly?
- **We can do syntax checking as we read integer values**
  - Skip over spaces and newline characters
  - Read digits until a non-digit is found
  - Calculate value of integer from digits read
  - Print an error message if something strange is read

# ASCII TO INTEGER CONVERSION

**// Sample program for reading integers**

```
int Num = 0;  
char Ch = ' ';
```

**// Skip over spaces and newlines**

```
while ((Ch == ' ') || (Ch == '\n'))  
    cin.get(Ch) ;  
if ( !cin.eof() )  
    cin.unget(Ch);
```

This will undo the last  
get command so we  
can read first digit of  
number below



# ASCII TO INTEGER CONVERSION

**// Read characters until non-digit is read**

```
while (cin.get(Ch) && (Ch >= '0') && (Ch <= '9'))
```


```
    Num = Num * 10 + int(Ch) - int('0');
```

```
if (!cin.eof())
```

```
    cin.unget(Ch);
```

**// Print value of integer**

```
cout << "Num = " << Num << endl;
```



This calculation makes use of the fact that ASCII codes for digits 0..9 are sequential



# INTEGER TO ASCII CONVERSION

- In most applications, we can simply use “`cout << num`” to output an integer value from a variable
  - What if we want to do this conversion ourselves?
  - It is easy to output digits in reverse order
  - It is harder to output digits in correct order
- How can we output integer digits in reverse order?
  - Use **modulo** operator to calculate least significant digit
  - Convert digit into corresponding ascii character
  - Print this ascii character to the output
  - Use **division** to remove least significant digit
  - Repeat until there are no more digits to output

# INTEGER TO ASCII CONVERSION

Example:

$\text{num} = 6324$

$\text{digit} = \text{num} \% 10 = 4$

$\text{num} = \text{num} / 10 = 632$

$\text{digit} = \text{num} \% 10 = 2$

$\text{num} = \text{num} / 10 = 63$

$\text{digit} = \text{num} \% 10 = 3$

$\text{num} = \text{num} / 10 = 6$

$\text{digit} = \text{num} \% 10 = 6$

$\text{num} = \text{num} / 10 = 0$



Notice that we  
calculated the least  
significant digits of num  
in the order 4 2 3 6

# INTEGER TO ASCII CONVERSION

**// Output ascii digits of integer in reverse order**

```
int Num = 4213;
```

```
char Ch;
```

**// Loop to calculate ascii digits**

```
while (Num != 0)
```

```
{
```


```
    Ch = char(Num % 10 + '0');
```

```
    Num = Num / 10;
```

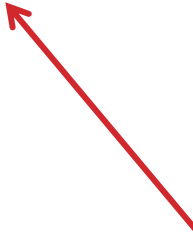
```
    cout.put(Ch);
```

```
}
```

Calculate the character  
representing the least  
significant digit



We can convert the integer to  
**octal** representation by using 8  
instead of 10 in these steps



# INTEGER TO ASCII CONVERSION

- **How can we output integer digits in correct order?**
  - Use modulo operator to calculate least significant digit
  - Convert digit into corresponding ascii character
  - Save this ascii character in an **array** of characters
  - Use division to remove least significant digit
  - Repeat until there are no more digits
  - Finally, loop over the array of characters in reverse order and output the digits in the correct order

# INTEGER TO ASCII CONVERSION

Example:

$\text{num} = 6324$

$\text{digit} = \text{num} \% 10 = 4$

$\text{num} = \text{num} / 10 = 632$

$\text{digit} = \text{num} \% 10 = 2$

$\text{num} = \text{num} / 10 = 63$

$\text{digit} = \text{num} \% 10 = 3$

$\text{num} = \text{num} / 10 = 6$

$\text{digit} = \text{num} \% 10 = 6$

$\text{num} = \text{num} / 10 = 0$

Array:

4

42

423

4236



We can print this  
array in reverse  
order to output  
6324

# INTEGER TO ASCII CONVERSION

**// Output ascii digits of integer in correct order**

```
int Num = 3901;
```

```
char Ch[10];
```

```
int Pos = 0;
```

**// Loop to calculate ascii digits**

```
while (Num != 0)
```

```
{
```


```
    Ch[Pos] = char(Num % 10 + '0');
```

```
    Num = Num / 10;
```


```
    Pos++;
```

```
}
```

This will save ascii  
digits in an array to  
be output later



We fill the array from left  
to right as we find digits



# INTEGER TO ASCII CONVERSION

**// Loop to print ascii digits in correct order**

```
while (Pos > 0)
```


```
{
```

```
    Pos--;
```

```
    cout.put(Ch[Pos]);
```

```
}
```

We loop from right to left down the array to print characters in the correct order



# SUMMARY

- **In this section, we learned more about C++ streams**
  - Additional properties of standard I/O streams
    - End of file detection
  - Single character I/O commands
    - Put, get, unget, peek, eof
  - ASCII to integer conversion
    - Character by character input
  - Integer to ASCII conversion
    - Character by character output



# **STREAMS AND FILES**

**PART 2**

**INPUT FILES**

# INPUT FILES

- **Reading program input from a file has several advantages**
  - We can input very large amounts of data
  - We can save this information long term in file system
  - We can read / edit this data using a text editor
  - We can process data created by another program
- **C++ has provided support for file input**
  - Add `#include <fstream>` at top of program
  - Use the `ifstream` object for program input

# INPUT FILES

- **To read data from an ASCII input file we must**
  - Declare object of ifstream class
    - Eg: ifstream din;
  - Open the input file
    - Eg: din.open("input.txt");
    - Or: ifstream din ("input.txt");
  - Check if open was successful
    - Eg: if (din.fail()) cout << "Error opening file\n";
    - ~~▪ Or: if (!din) cout << "Error opening file\n";~~
  - Read data from the input file
    - Eg: din >> variable;
  - Close the input file
    - Eg: din.close();

# READING INTEGERS

- Program to read and total all integer values in a file

```
ifstream din; ← This declares an input  
int num = 0;      stream called "din"  
int total = 0;
```

```
    din.open("numbers.txt");  
    if ( din.fail() )  
        cout << "Error opening file.\n";  
    else  
    {  
        while (din >> num)  
            total += num;  
        cout << "total=" << total << endl;  
        din.close();  
    }
```

# READING INTEGERS

- Program to read and total all integer values in a file

```
ifstream din;  
int num = 0;  
int total = 0;
```

This will open a file  
called numbers.txt in  
the current directory



```
    din.open("numbers.txt");  
    if ( din.fail() )  
        cout << "Error opening file.\n";  
    else  
    {  
        while (din >> num)  
            total += num;  
        cout << "total=" << total << endl;  
        din.close();  
    }
```

# READING INTEGERS

- Program to read and total all integer values in a file

```
ifstream din;  
int num = 0;  
int total = 0;
```


```
din.open("numbers.txt");  
if ( din.fail() )  
    cout << "Error opening file.\n";  
else  
{  
    while (din >> num)  
        total += num;  
    cout << "total=" << total << endl;  
    din.close();  
}
```

← This will report an error if din.open fails

# READING INTEGERS

- Program to read and total all integer values in a file

```
ifstream din;  
int num = 0;  
int total = 0;
```

```
din.open("numbers.txt");  
if ( din.fail() )  
    cout << "Error opening file.\n";  
else  
{  
    while (din >> num)   
        total += num;  
    cout << "total=" << total << endl;  
    din.close();  
}
```

This loop will read integers until the end of file is reached

# READING INTEGERS

- Program to read and total all integer values in a file

```
ifstream din;  
int num = 0;  
int total = 0;
```

```
din.open("numbers.txt");  
if ( din.fail() )  
    cout << "Error opening file.\n";  
else  
{  
    while (din >> num)  
        total += num;  
    cout << "total=" << total << endl;  
    din.close();  
}
```

When we are finished  
reading data we  
should close the file



# READING INTEGERS

- **Sample input.txt file (all values on one line)**

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

- **Sample input.txt file (five values per line)**

```
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
```

- **It does not matter how this input file is formatted because the `din >> num` command will skip over white space between the integer values**

# READING MIXED DATA

- To read variables with different types from an input file we need to know what **order** the data is stored in the file
- For example, if we want to read three pieces of student information it could be stored in six different ways!
  - student\_id      last\_name      GPA
  - student\_id      GPA      last\_name
  - last\_name      student\_id      GPA
  - last\_name      GPA      student\_id
  - GPA      student\_id      last\_name
  - GPA      last\_name      student\_id

# READING MIXED DATA

- **Assume that the input file is in the following format**
  - One student record per line in the file
  - Data order: `student_id` `last_name` `GPA`
- **The goal of our program is to read the input file and print out the names of all students with `GPA >= 3.0`**
  - Open input file
  - Loop reading student data
    - Check student GPA
    - Print selected student names

# READING MIXED DATA

- Program to read and process student data

```
ifstream din;  
int ID;  
string Name;  
float GPA;  
din.open("student.txt");  
if ( !din.fail() )  
{  
    while (din >> ID >> Name >> GPA)  
        if (GPA > 3.0) cout << Name << endl;  
    din.close();  
}
```

This will open a file  
called student.txt in  
the current directory;




# READING MIXED DATA

- Program to read and process student data

```
ifstream din;  
int ID;  
string Name;  
float GPA;  
din.open("student.txt");  
if ( !din.fail() )  
{  
    while (din >> ID >> Name >> GPA)  
        if (GPA > 3.0) cout << Name << endl;  
    din.close();  
}
```

Read three pieces of  
student data from the  
input file in **this** order



# READING MIXED DATA

- Program to read and process student data

```
ifstream din;  
int ID;  
string Name;  
float GPA;  
din.open("student.txt");  
if ( !din.fail() )  
{  
    while (din >> ID >> Name >> GPA)  
        if (GPA > 3.0) cout << Name << endl;  
    din.close();  
}
```

← Print selected student name

# READING MIXED DATA

- **Sample student.txt file**

```
123   Smith   3.5
234   Jones   2.7
345   Brown   3.1
456   Taylor  2.3
...
```

- **Advantages of this input file format**


- Puts student data in correct order for this program
- Keeps all information about one student on one line
- Easier to read / edit than one student variable per line

# READING MIXED DATA

```
string filename;  
cout << "Enter student filename: ";  
cin >> filename;
```

```
ifstream din;  
din.open( filename.c_str() );
```

```
if ( din.fail() )  
    cout << "Error: could not open " << filename << endl;
```



In this program we prompt the user to enter the name of the student input file




# READING MIXED DATA

```
string filename;  
cout << "Enter student filename: ";  
cin >> filename;
```

```
ifstream din;  
din.open( filename.c_str() );
```

```
if ( din.fail() )  
    cout << "Error: could not open " << filename << endl;
```



The `c_str()` command converts a string variable into a `cstring` variable, which is what `open` expects as an input parameter

# READING MIXED DATA


```
string filename;  
cout << "Enter student filename: ";  
cin >> filename;
```

```
ifstream din;  
din.open( filename.c_str() );
```

```
if ( din.fail() )
```

```
    cout << "Error: could not open " << filename << endl;
```

If the user enters an  
invalid file name we  
print out an error msg



# READING MIXED DATA

...

else

{

int ID[10];

string Name[10];

float GPA[10];

int i = 0;

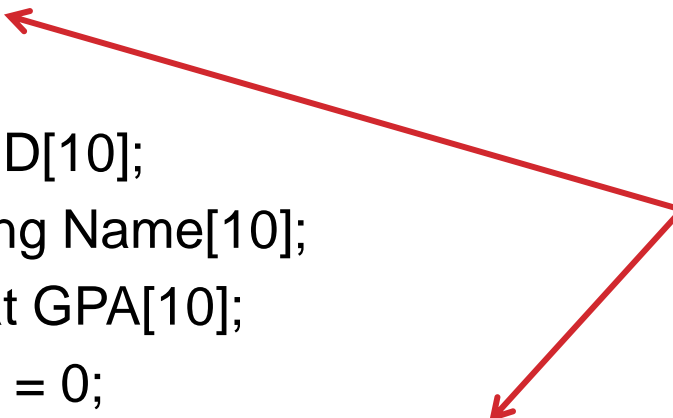
while (din >> ID[i] >> Name[i] >> GPA[i++])

if (GPA > 3.0) cout << Name[i-1] << endl;

din.close();

}

We only read the student data if the file open was successful



# SUMMARY

- **In this section described the C++ syntax for file input**
  - How to declare an ifstream object
  - How to open a file
  - How to check for file open errors
  - How to read from the file
  - How to close the file
- **Key concept: The program that reads the file must know the **format** of the input file in advance**
  - Otherwise values will be read into the wrong variables

# **STREAMS AND FILES**

**PART 3**

**OUTPUT FILES**

# OUTPUT FILES

- **Writing program output into a file has several advantages**
  - We can output very large amounts of data
  - We can save this information long term in file system
  - We can read / edit this data using a text editor
  - We can process this data using another program
- **C++ has provided support for file output**
  - Add `#include <fstream>` at top of program
  - Use the ofstream object for program output

# OUTPUT FILES

- **To write data into an ASCII output file we must**
  - Declare object of ofstream class
    - Eg: ofstream dout;
  - Open the output file
    - Eg: dout.open("output.txt");
  - Check if open was successful
    - Eg: if ( dout.fail() ) cout << "Error opening file\n";
  - Write data into the output file
    - Eg: dout << variable;
  - Close the output file
    - Eg: dout.close();

# WRITING INTEGERS

- Program to output the times table up to 12x12

```
ofstream dout;
dout.open("output.txt");
if (dout.fail()) return;
```

This declares an output stream called “dout”

```
for (int row = 1; row <= 12; row++)
{
    for (int col = 1; col <= 12; col++)
        dout << row * col << " ";
    dout << endl;
}
dout.close();
```



# WRITING INTEGERS

- Program to output the times table up to 12x12

```
ofstream dout;  
dout.open("output.txt");  
if (dout.fail()) return;
```



This will create a new file called output.txt in the current directory


```
for (int row = 1; row <= 12; row++)  
{  
    for (int col = 1; col <= 12; col++)  
        dout << row * col << " ";  
    dout << endl;  
}  
dout.close();
```

# WRITING INTEGERS

- Program to output the times table up to 12x12

```
ofstream dout;  
dout.open("output.txt");  
if (dout.fail()) return;
```

```
for (int row = 1; row <= 12; row++)  
{  
    for (int col = 1; col <= 12; col++)  
        dout << row * col << " ";  
    dout << endl;  
}  
dout.close();
```



This will output 12  
integer values per line  
separated by spaces

# WRITING INTEGERS

- **Contents of output.txt file**

```
1 2 3 4 5 6 7 8 9 10 11 12
2 4 6 8 10 12 14 16 18 20 22 24
3 6 9 12 15 18 21 24 27 30 33 36
4 8 12 16 20 24 28 32 36 40 44 48
5 10 15 20 25 30 35 40 45 50 55 60
6 12 18 24 30 36 42 48 54 60 66 72
7 14 21 28 35 42 49 56 63 70 77 84
8 16 24 32 40 48 56 64 72 80 88 96
9 18 27 36 45 54 63 72 81 90 99 108
10 20 30 40 50 60 70 80 90 100 110 120
11 22 33 44 55 66 77 88 99 110 121 132
12 24 36 48 60 72 84 96 108 120 132 144
```

# WRITING MIXED DATA

- **When we write variables with different data types to a file we need to make the format easy to read**
  - Group data that belongs together on one line
  - Put data fields in a natural order
  - Put variable length fields at end of the line
- **Example: Writing student information to a file**
  - Assume student data is stored in three arrays
  - Output order: ID GPA Name
  - One student record per line in the output file

# WRITING MIXED DATA

- Program to output student data to a file

```
ofstream dout;
```

```
dout.open("student.txt");
```

```
if (dout.fail() ) return;
```

This will open a file  
called student.txt in  
the current directory



```
for (int i=0; i<count; i++)
```

```
    dout << ID[i] << " "
```

```
        << GPA[i] << " "
```

```
        << Name[i] << endl;
```

```
dout.close();
```

# WRITING MIXED DATA

- Program to output student data to a file

```
ofstream dout;  
dout.open("student.txt");  
if (dout.fail() ) return;  
  
for (int i=0; i<count; i++)  
    dout << ID[i] << " "  
        << GPA[i] << " "  
        << Name[i] << endl;  
dout.close();
```

← This will quietly exit  
this function if the file  
open fails

# WRITING MIXED DATA

- Program to output student data to a file

```
ofstream dout;  
dout.open("student.txt");  
if (dout.fail() ) return;
```

```
for (int i=0; i<count; i++)  
    dout << ID[i] << " "  
        << GPA[i] << " "  
        << Name[i] << endl;  
dout.close();
```

← This will loop over three arrays that contain student information and write data to the file

# WRITING MIXED DATA

- Program to output student data to a file

```
ofstream dout;  
dout.open("student.txt");  
if (dout.fail() ) return;  
  
for (int i=0; i<count; i++)  
    dout << ID[i] << " \t"  
        << GPA[i] << "\t"  
        << Name[i] << endl;  
dout.close();
```

This will close the  
output file after all  
data has been written





# WRITING MIXED DATA

- **Sample student.txt file**

```
123    3.5    Smith
234    2.7    Jones
345    3.1    Jean Claude Van Dam
456    2.3    Cher
...
```

Use `getline(din, name)`  
to read the name

- **Notice that this output format has data in a **different order** than our previous student input file format**
  - We can NOT read this student.txt file using our previous student input program
  - We must change either the input format or output format so they match each other

# **STREAMS AND FILES**

## **PART 4**

### **EXAMPLE: COPYING A FILE**


# COPYING A FILE

- **This example will demonstrate how to make an exact copy of an input file**
  - We will write a function, `copyFile`, that is given the name of the input file to copy and the name of the output file to create
  - `copyFile` will loop until end of the input file reading one character and then writing it to the output file
  - We will read with **`get`** because `>>` skips whitespace and we want to copy the whitespace characters too

# COPYING A FILE

```
int main()
{
    if (!copyFile("input.txt", "output.txt"))
        cerr << "Copy file failed\n";
    return 0;
}
```

If the copyFile function fails then print an error message to the cerr “standard error” stream



# FUNCTION TO COPY FILE

```
bool copyFile (const string inFile, const string outFile)
```

```
{  
    ifstream din(inFile.c_str()); ← Declare and open din for  
    ofstream dout(outFile.c_str()); reading and dout for writing  
    char Ch = ' ';  
    bool Success = false;  
  
    if (din.fail()) cerr << "Error, file " << inFile << " did not exist.\n" ;  
    else if (dout.fail()) cerr << "Error, could not open " << outFile << endl;  
    else  
    {  
        while (din.get(Ch))  
            dout.put(Ch);  
        din.close();  
        dout.close();  
        Success = true;  
    }  
    return Success;  
}
```

# FUNCTION TO COPY FILE

```
bool copyFile (const string inFile, const string outFile)
```

```
{  
    ifstream din(inFile.c_str());  
    ofstream dout(outFile.c_str());  
    char Ch = ' ';  
    bool Success = false;  
  
    if (din.fail()) cerr << "Error, file " << inFile << " did not exist.\n" ;  
    else if (dout.fail()) cerr << "Error, could not open " << outFile << endl;  
    else  
    {  
        while (din.get(Ch))  
            dout.put(Ch);  
        din.close();  
        dout.close();  
        Success = true;  
    }  
    return Success;  
}
```

← Declare Ch to hold a character read from the file and a status flag, Success, initialized to false

# FUNCTION TO COPY FILE

```
bool copyFile (const string inFile, const string outFile)
```

```
{  
    ifstream din(inFile.c_str());  
    ofstream dout(outFile.c_str());  
    char Ch = ' ';  
    bool Success = false;
```

If either file did not open properly, print an error message to cerr



```
    if (din.fail()) cerr << "Error, file " << inFile << " did not exist.\n" ;  
    else if (dout.fail()) cerr << "Error, could not open " << outFile << endl;  
    else  
    {  
        while (din.get(Ch))  
            dout.put(Ch);  
        din.close();  
        dout.close();  
        Success = true;  
    }  
    return Success;  
}
```

# FUNCTION TO COPY FILE

```
bool copyFile (const string inFile, const string outFile)
{
    ifstream din(inFile.c_str());
    ofstream dout(outFile.c_str());
    char Ch = ' ';
    bool Success = false;

    if (din.fail()) cerr << "Error, file " << inFile << " did not exist.\n" ;
    else if (dout.fail()) cerr << "Error, could not open " << outFile << endl;
    else
    {
        while (din.get(Ch))           ← Read a character until end
            dout.put(Ch);             of file using get to capture
        din.close();                  whitespace chars too
        dout.close();
        Success = true;
    }
    return Success;
}
```



# FUNCTION TO COPY FILE

```
bool copyFile (const string inFile, const string outFile)
{
    ifstream din(inFile.c_str());
    ofstream dout(outFile.c_str());
    char Ch = ' ';
    bool Success = false;


    if (din.fail()) cerr << "Error, file " << inFile << " did not exist.\n" ;
    else if (dout.fail()) cerr << "Error, could not open " << outFile << endl;
    else
    {
        while (din.get(Ch))
            dout.put(Ch);
        din.close();
        dout.close();
        Success = true;
    }
    return Success;
}
```

← Write each character to dout using put (<< would work too)

# FUNCTION TO COPY FILE

```
bool copyFile (const string inFile, const string outFile)
{
    ifstream din(inFile.c_str());
    ofstream dout(outFile.c_str());
    char Ch = ' ';
    bool Success = false;

    if (din.fail()) cerr << "Error, file " << inFile << " did not exist.\n" ;
    else if (dout.fail()) cerr << "Error, could not open " << outFile << endl;
    else
    {
        while (din.get(Ch))
            dout.put(Ch);
        din.close();
        dout.close();
        Success = true;
    }
    return Success;
}
```



Close both files and set  
Success variable to true

# FUNCTION TO COPY FILE

```
bool copyFile (const string inFile, const string outFile)
{
    ifstream din(inFile.c_str());
    ofstream dout(outFile.c_str());
    char Ch = ' ';
    bool Success = false;


    if (din.fail()) cerr << "Error, file " << inFile << " did not exist.\n" ;
    else if (dout.fail()) cerr << "Error, could not open " << outFile << endl;
    else
    {
        while (din.get(Ch))
            dout.put(Ch);
        din.close();
        dout.close();
        Success = true;
    }
    return Success;
}
```

← Return the value of Success (true or false)

# FUNCTION TO COPY FILE

```
bool copyFile (const string inFile, const string outFile)
{
    ifstream din(inFile.c_str());
    ofstream dout(outFile.c_str());
    char Ch = ' ';
    bool Success = false;

    if (din.fail()) cerr << "Error, file " << inFile << " did not exist.\n" ;
    else if (dout.fail()) cerr << "Error, could not open " << outFile << endl;
    else
    {
        while (din >> Ch)
            dout << Ch;
        din.close();
        dout.close();
        Success = true;
    }
    return Success;
}
```



This will NOT copy the spaces into output file

# **STREAMS AND FILES**

## **PART 4**

### **EXAMPLE: FILLING AN ARRAY**

# FILLING AN ARRAY

- **This example will demonstrate how we can read data from a file and store this information in an array**
  - We will write a function called `fillArray`, that is given an input file, an array to hold the data, and the array size
  - The main program will open the input file, call the `fillArray` function, and close the input file
  - The `fillArray` function will read data values from the input file one by one, and store these values in the array
  - The function will return the number of data values read into the array (the rest of the array will be left empty)

# FILLING AN ARRAY

```
#include <iostream>
```

```
#include <fstream>
```

```
int main()
```

```
{
```

```
    const int MAXSTUDENTS = 100;
```

```
    float studentArray[MAXSTUDENTS];
```

```
    int numStudents = 0;
```

```
    ifstream din("students.txt");
```

```
    if (din.fail())
```

```
        cout << "Could not open students.txt.\n";
```

```
    else
```

```
    {
```

```
        numStudents = fillArray (din, studentArray, MAXSTUDENTS));
```

```
        for (int i = 0; i < numStudents; i++)
```

```
            cout << studentArray[i] << " ";
```

```
        cout << endl;
```


```
        din.close();
```

```
    }
```

```
    return 0;
```

```
}
```

Declare a fixed size  
array to hold the data



# FILLING AN ARRAY

```
#include <iostream>
```

```
#include <fstream>
```

```
int main()
```

```
{
```

```
    const int MAXSTUDENTS = 100;
```

```
    float studentArray[MAXSTUDENTS];
```

```
    int numStudents = 0;
```

```
    ifstream din("students.txt");
```

← Declare din and open  
"students.txt" for reading

```
    if (din.fail())
```

```
        cout << "Could not open students.txt.\n";
```

```
    else
```

```
    {
```

```
        numStudents = fillArray (din, studentArray, MAXSTUDENTS));
```

```
        for (int i = 0; i < numStudents; i++)
```

```
            cout << studentArray[i] << " ";
```

```
        cout << endl;
```

```
        din.close();
```

```
    }
```

```
    return 0;
```

```
}
```



# FILLING AN ARRAY

```
#include <iostream>
```

```
#include <fstream>
```

```
int main()
```

```
{
```

```
    const int MAXSTUDENTS = 100;
```

```
    float studentArray[MAXSTUDENTS];
```

```
    int numStudents = 0;
```

```
    ifstream din("students.txt");
```

```
    if (din.fail())
```

```
        cout << "Could not open students.txt.\n";
```

```
    else
```

```
    {
```

```
        numStudents = fillArray (din, studentArray, MAXSTUDENTS,));
```

```
        for (int i = 0; i < numStudents; i++)
```

```
            cout << studentArray[i] << " ";
```

```
        cout << endl;
```

```
        din.close();
```

```
    }
```

```
    return 0;
```

```
}
```

Call the function to fill the array from the opened file



# FILLING AN ARRAY

```
#include <iostream>
```

```
#include <fstream>
```

```
int main()
```

```
{
```

```
    const int MAXSTUDENTS = 100;
```

```
    float studentArray[MAXSTUDENTS];
```

```
    int numStudents = 0;
```

```
    ifstream din("students.txt");
```

```
    if (din.fail())
```

```
        cout << "Could not open students.txt.\n";
```

```
    else
```

```
    {
```

```
        numStudents = fillArray (din, studentArray, MAXSTUDENTS,);
```

```
        for (int i = 0; i < numStudents; i++)
```

```
            cout << studentArray[i] << " ";
```

```
        cout << endl;
```

```
        din.close();
```

```
    }
```

```
    return 0;
```

```
}
```

← Print the array back out to check that it was filled properly. Use numStudents to avoid unused elements

# FILLING AN ARRAY

```
#include <iostream>
```

```
#include <fstream>
```

```
int main()
```

```
{
```

```
    const int MAXSTUDENTS = 100;
```

```
    float studentArray[MAXSTUDENTS];
```

```
    int numStudents = 0;
```

```
    ifstream din("students.txt");
```

```
    if (din.fail())
```

```
        cout << "Could not open students.txt.\n";
```

```
    else
```

```
    {
```

```
        numStudents = fillArray (din, studentArray, MAXSTUDENTS,);
```

```
        for (int i = 0; i < numStudents; i++)
```

```
            cout << studentArray[i] << " ";
```

```
        cout << endl;
```

```
        din.close();
```



Close the file  
when finished

```
    }
```

```
    return 0;
```

```
}
```

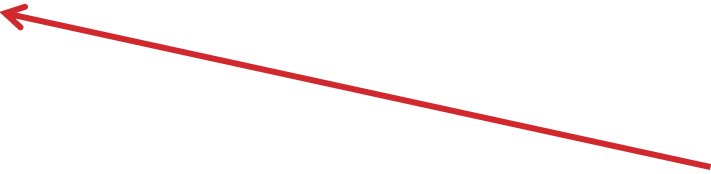
# FUNCTION TO FILL ARRAY

// Fills the array until end of file is hit

```
int fillArray (ifstream &din, float array[], const int size)
```

```
{  
    float tmp;  
    int numRead = 0;  
  
    din >> tmp;  
    while (!din.eof() && numRead < size)  
    {  
        array[numRead] = tmp;  
        numRead++;  
        din >> tmp;  
    }  
    if (!din.eof())  
        cerr << "Could not fill all the data in the array.\n";  
    return numRead;  
}
```

Always pass streams  
by reference because  
reading/writing  
changes the stream by  
adding/removing data



# FUNCTION TO FILL ARRAY

// Fills the array until end of file is hit

```
int fillArray (ifstream &din, float array[], const int size)
```

```
{
```

```
    float tmp;
```

```
    int numRead = 0;
```

```
    din >> tmp;
```



Read into a temporary variable since this read will fail at eof

```
    while (!din.eof() && numRead < size)
```

```
    {
```

```
        array[numRead] = tmp;
```

```
        numRead++;
```

```
        din >> tmp;
```

```
    }
```

```
    if (!din.eof())
```

```
        cerr << "Could not fill all the data in the array.\n";
```


```
    return numRead;
```

```
}
```

# FUNCTION TO FILL ARRAY

// Fills the array until end of file is hit

```
int fillArray (ifstream &din, float array[], const int size)
```

```
{  
    float tmp;  
    int numRead = 0;  
  
    din >> tmp;  
    while (!din.eof() && numRead < size)   
    {  
        array[numRead] = tmp;  
        numRead++;  
        din >> tmp;  
    }  
    if (!din.eof())  
        cerr << "Could not fill all the data in the array.\n";  
    return numRead;  
}
```

Loop while we have not  
read the eof marker  
AND we still have  
space in the array


# FUNCTION TO FILL ARRAY

// Fills the array until end of file is hit

```
int fillArray (ifstream &din, float array[], const int size)
```

```
{  
    float tmp;  
    int numRead = 0;  
  
    din >> tmp;  
    while (!din.eof() && numRead < size)  
    {  
        array[numRead] = tmp;  
        numRead++;  
        din >> tmp;  
    }  
    if (!din.eof())  
        cerr << "Could not fill all the data in the array.\n";  
    return numRead;  
}
```


The previous read  
succeeded, store that  
value in the array;  
update the count of  
values read



# FUNCTION TO FILL ARRAY

```
// Fills the array until end of file is hit
int fillArray (ifstream &din, float array[], const int size)
{
    float tmp;
    int numRead = 0;

    din >> tmp;
    while (!din.eof() && numRead < size)
    {
        array[numRead] = tmp;
        numRead++;
        din >> tmp;
    }
    if (!din.eof())
        cerr << "Could not fill all the data in the array.\n";
    return numRead;
}
```



Read the next value  
from the input file



# FUNCTION TO FILL ARRAY

// Fills the array until end of file is hit

```
int fillArray (ifstream &din, float array[], const int size)
```

```
{
```

```
    float tmp;
```

```
    int numRead = 0;
```

```
    din >> tmp;
```

```
    while (!din.eof() && numRead < size)
```

```
    {
```

```
        array[numRead] = tmp;
```

```
        numRead++;
```

```
        din >> tmp;
```

```
    }
```

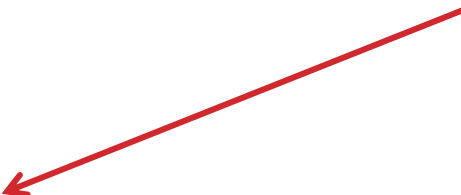
```
    if (!din.eof())
```

```
        cerr << "Could not fill all the data in the array.\n";
```

```
    return numRead;
```

```
}
```

If we are not at eof, then there are unread values (we stopped because the array was full)



# FUNCTION TO FILL ARRAY

// Fills the array until end of file is hit

```
int fillArray (ifstream &din, float array[], const int size)
```

```
{
```

```
    float tmp;
```

```
    int numRead = 0;
```

```
        din >> tmp;
```

```
        while (!din.eof() && numRead < size)
```

```
        {
```

```
            array[numRead] = tmp;
```

```
            numRead++;
```

```
            din >> tmp;
```

```
        }
```

```
        if (!din.eof())
```

```
            cerr << "Could not fill all the data in the array.\n";
```

```
    return numRead;
```

```
}
```



Return the number of  
values read into the array.

# FUNCTION TO FILL ARRAY

// Fills the array until end of file is hit

```
int fillArray (ifstream &din, float array[], const int size)
```

```
{
```

```
    float tmp;
```

```
    int numRead = 0;
```

```
    while ((din >> tmp) && (numRead < size))
```

```
        array[numRead++] = tmp;
```



Here is a more compact  
version of the read loop.

```
    if (!din.eof())
```

```
        cerr << "Could not fill all the data in the array.\n";
```

```
    return numRead;
```

```
}
```

# SOFTWARE ENGINEERING TIPS

- **Remember to put spaces or tabs between data values**
  - Otherwise your output data values will be unreadable
- **Be very careful when opening output files**
  - If you open a file that already exists, you will **erase** the original file and overwrite it with your output
  - This can be very bad, especially if you use the name of the input file (or your source code!) by accident

# SUMMARY

- **In this section described the C++ syntax for file output**
  - How to declare an ofstream object
  - How to open a file
  - How to check for file open errors
  - How to write to the file
  - How to close the file