Programming Design

C++ Strings, File I/O, and Header Files

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Applications of classes

- Let’s study two applications of classes.
  - C++ strings.
  - File input/output.
- Let’s also study a better way of managing a program (with classes).
  - Self-defined header files.

Outline

- **C++ Strings**
- File I/O
- Self-defined header files

C++ Strings: `string`

- There are two types of strings:
  - C string: the string represented by a character array with a `\0` at the end.
  - C++ string: the **class** `string` defined in `<string>`.
- A C++ string is more convenient and powerful than a C string.
- In the class `string`, there are:
  - A **member variable**, a pointer pointing to a dynamic character array.
  - Many **member functions**.
  - Many **overloaded operators**.

`string` declaration

- Let’s declare some C++ strings:

```
string myStr;
string yourStr = "your string";
string herStr(yourStr);
```

```
string::string();
string::string(const char* s);
string::string(const string& str);
```

  - `string` is a class defined in `<string>`.
  - `string` is not a C++ keyword.
  - `myStr` is an object.
  - Thanks to constructors!
- To use a C++ string, one does not need to worry about a **null character**.
  - Thanks to encapsulation!

`string` lengths

- We may use the member functions `length()` or `size()` to get the string length.
  - Just like `strlen()` for C strings.

```
string myStr;
string yourStr = "your string";
cout << myStr.length() << endl; // 0
cout << yourStr.size() << endl; // 11
```

```
size_t string::length() const;
size_t string::size() const;
```

- How long a string may be? Call `max_size()` to see:

```
string myStr;
cout << myStr.max_size() << endl;
// 4611686018427387897
```

```
size_t string::max_size() const;
```

`string` assignment

- C++ string **assignment** is easy and intuitive:

```
string myString = "my string";
string yourString = myString;
string herString;
herString = yourString = "a new string";
```

- We may also assign a C string to a C++ string.

```
char hisString[100] = "oh ya";
myString = hisString;
```

- Thanks to operator overloading!

`string` concatenation and indexing

- C++ strings can be **concatenated** with **+**.
  - Just like **`strcat()`** in C string.

```
string myStr = "my string ";
string yourStr = myStr;
string herStr;
herStr = myStr + yourStr;
  // "my string my string "
```

- String literals or C strings also work.
  - **`+=`** also works.

```
string s = "123";
char c[100] = "456";
string t = s + c;
string u = s + "789" + t;
```

- To access a character in a C++ string, use **`[]`**.

```
string myString = "my string";
char a = myString[0]; // m
```

- Thanks to operator overloading!

string input: getline()

- For **cin >>** to input into a C++ string, **white spaces** are still delimiters.
- To fix this, now we cannot use **cin.getline()**.
  - The first argument of **cin.getline()** must be a C string.
- We use a global function **getline()** defined in **<string>** instead:

```
string s;
getline(cin, s);
```

```
istream& getline(istream& is, string& str);
```

- By default, **getline()** stops when reading a newline character. We may specify the delimiter character we want:

```
string s;
getline(cin, s, '#');
```

```
istream& getline(istream& is, string& str, char delim);
```

- Note that there is **no length limitation**.

Substrings

- We may use **substr()** to get the **substring** of a string.

```
string string::substr(size_t pos = 0, size_t len = npos) const;
```

- **string::npos** is a static member variable indicating the maximum possible value of type **size_t**.
- As an example:

```
string s = "abcdef";
cout << s.substr(2, 3) << endl; // "cde"
cout << s.substr(2) << endl; // "cdef"
```

string finding

- We may use the member function **find()** to look for a string or character.
  - Just like **strstr()** and **strchr()** for C strings.

```
size_t find(const string& str, size_t pos = 0) const;
size_t find(const char* s, size_t pos = 0) const;
size_t find(char c, size_t pos = 0) const;
```

- This will return the beginning index of the argument, if it exists, or **string::npos** otherwise.

```
string s = "abcdefg";
if(s.find("bcd") != string::npos)
  cout << s.find("bcd"); // 1
```

`string` comparisons

- We may use **`>`**, **`>=`**, **`<`**, **`<=`**, **`==`**, **`!=`** to **compare** two C++ strings.
  - According to the alphabetical order.
  - Just like **`strcmp()`**.
- String literals or C strings also work.
  - As long as one side of the comparison is a C++ string, it is fine.
  - Thanks to operator overloading.
  - However, if none of the two sides is a C++ string, there will be an error.
- Look up these functions of string, and more, from books or websites.

Insertion, replacement, and erasing

- We may use **insert()**, **replace()**, and **erase()** to modify a string.

```
string& insert(size_t pos, const string& str);
string& replace(size_t pos, size_t len, const string& str);
string& erase(size_t pos = 0, size_t len = npos);
```

```
int main()
{
  cout << "01234567890123456789\n";
  string myStr = "Today is not my day.";
  myStr.insert(9, "totally "); // Today is totally not my day.
  myStr.replace(17, 3, "NOT"); // Today is totally NOT my day.
  myStr.erase(17, 4); // Today is totally my day.
  cout << myStr << endl;
  return 0;
}
```

C++ strings to/from other types

- A C++ string can be easily converted to other types.
  - To convert a C++ string to a C string, use the member function **`c_str()`**.
  - To convert a C++ string to a number, use the global functions **`stoi()`**, **`stof()`**, **`stod()`**, etc.
  - To convert a number to a C++ string, use the global functions **`to_string()`**.
- Check out these functions by yourself!

C++ strings for Chinese characters

- Nowadays, C and C++ strings all accept **Chinese characters**.
- Different environment may use different encoding systems (Big-5, UTF-8, etc.)
  - Most of them use **two bytes** to represent one Chinese character.

```
int main()
{
  string s = "大家好";
  cout << s << endl; // 大家好

  char c[100] = "喔耶";
  cout << c << endl; // 喔耶

  return 0;
}
```

```
int main()
{
  string s = "大家好";
  cout << s[1] << endl; // j

  char c[100] = "喔耶";
  cout << c + 2 << endl; // 耶

  return 0;
}
```

C++ strings for Chinese characters

- Functions in **<string>** all work for Chinese strings.
- However, many of them simply treat elements as **separated char variables**.
- As an example, let’s reverse a C++ string:

```
int main()
{
  string s = "12345";
  int n = s.length(); // 5
  string t = s;
  for(int i = 0; i < n; i++)
    t[n - i - 1] = s[i]; // good
  cout << t << endl; // 54321
  return 0;
}
```

```
int main()
{
  string s = "大家好";
  int n = s.length(); // 6
  string t = s;
  for(int i = 0; i < n; i++)
    t[n - i - 1] = s[i]; // bad
  cout << t << endl; // n地屐
  return 0;
}
```

C++ strings for Chinese characters

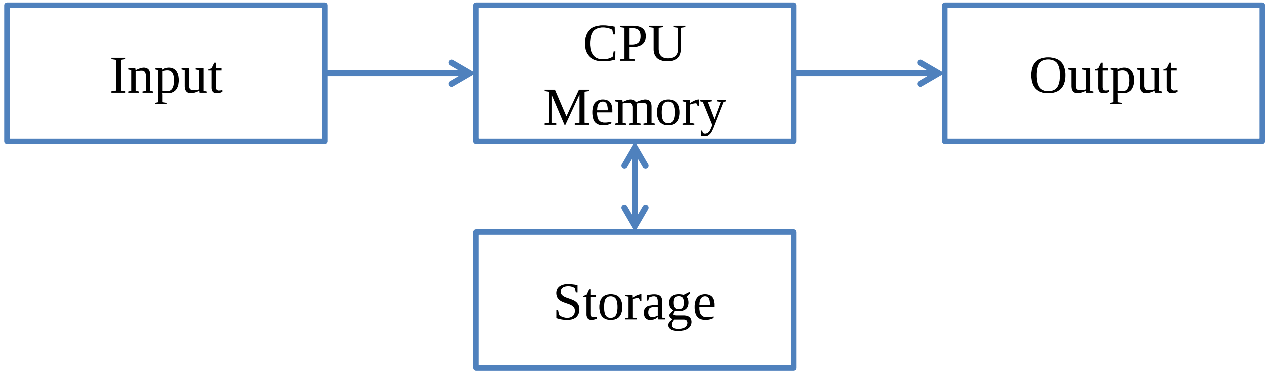
- For a C++ string with Chinese contents, the following program works:

```
int main()
{
  string s = "大家好";
  int n = s.length(); // 6
  string t = s;
  for(int i = 0; i < n - 1; i = i + 2)
  {
    t[n - i - 2] = s[i];
    t[n - i - 1] = s[i + 1];
  } // good
  cout << t << endl; // 好家大
  return 0;
}
```

Outline

- C++ Strings
- **File I/O**
- Self-defined header files

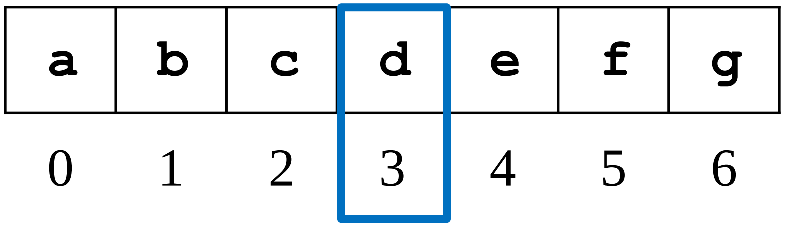
File I/O



- The **von Neumann architecture**:
- With the techniques of **file input/output** (file I/O), we will read data from and store data to files in the **hard discs**.
  - So that the results can still be kept **after** the program **terminates**.
- We will focus on **plain-text files**.
  - Those files that can be directly edited with Notepad on MS Windows.

A plain-text file

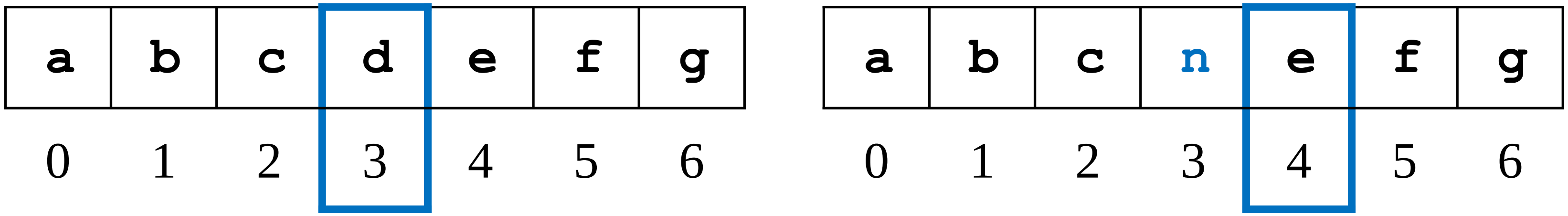
- Files store data.
  - A plain-text file stores **characters**.
  - A MS Word document stores characters and **format** information.
  - A bitmap file stores **color** codes.
- How are characters stored in a plain-text files?
  - Each character has its own **position**.
  - For each opened file, there is a **position pointer** indicating the **current reading/writing position**.
  - To control the reading/writing operations, we control the position pointer.



a	b	c	d	e	f	g
0	1	2	3	4	5	6

Writing to a file

- The first character is stored at **position 0**.
- In general, once a character is written to a file:
  - The character replaces the old character at the **current** position.
  - The position pointer moves to the **next** position (from *i* to *i* + 1).
- When a character `n` is written to this file:



a	b	c	d	e	f	g
0	1	2	3	4	5	6

a	b	c	n	e	f	g
0	1	2	3	4	5	6

File streams

- In C++, input and output activities are managed in **streams**.
  - E.g., data may flow from **`cin`** or into **`cout`**.
- To replace the console and keyboard by files, in C++ we create **`ifstream`** and **`ofstream`** objects.
- **`ifstream`** and **`ofstream`** are classes defined in **`<fstream>`**.
  - They can be used to create input/output file stream objects.
  - Simply imagine those objects as source/target files!

Output file streams

- To open and close an **output file stream**:

```
ofstream file object;
file object.open(file name);
// ...
file object.close();
```

```
ofstream myFile;
myFile.open("temp.txt");
// ...
myFile.close();
```

  - `open()` and `close()` are **public member functions**.
  - *file name* can be a C or C++ string.
- Thanks to encapsulation, we do not care about:
  - Whether there is a member variables storing the file name.
  - How `open()` and `close()` are implemented.

Writing to an output file stream

- To write to an output file stream, we may use **`<<`**.

```
ofstream myFile;
myFile.open("temp.txt");
myFile << "1 abc\n &%^ " << 123.45;
myFile.close();
```

  - **`<<`** has been **overloaded** for the class **`ofstream`**.
  - It returns **`ofstream&`** for concatenated output streams.
- What if we replace **`myFile`** by **`cout`** in the third statement?
- The second argument of **`<<`** can be of any basic data type.
  - What if we want to put a **`MyVector`** object as the second argument?

Options for an output file stream

- An **open mode** can be set when we open an output file stream.

```
ofstream file object;
file object.open(file name, option);
// ...
file object.close();
```

  - `ios::out` (default): The window starts at location 0; remove existing data.
  - `ios::app`: The window starts at the end; never modify existing data.
  - `ios::ate`: The window starts at the end; can modify existing data.
- **`ios`** is a class; **`out`**, **`app`**, and **`ate`** are **public static variables**.

Constructors and other members

- The class **ofstream** also provides **constructors**:

```
ofstream file object(file name, option);
```

```
ofstream file object(file name);
```

```
ofstream myFile("temp.txt");
myFile << "1 abc\n &%^ " << 123.45;
myFile.close();
```

  - Regardless of the extension name, we are creating/opening a plain text file.
- **ofstream** provides other member functions.
  - E.g., **put(char c)** writes the character **c** into the file.

Example

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

int main()
{
  ofstream scoreFile
    ("temp.txt", ios::out);
  char name[20] = {0};
  int score = 0;
  char notFin = 0;
  bool con = true;
```

```
  if(!scoreFile)
    exit(1);
  while(con)
  {
    cin >> name >> score;
    scoreFile << name << " " << score << "\n";
    cout << "Continue (Y/N)? ";
    cin >> notFin;
    con = ((notFin == 'Y') ? true : false);
  }
  scoreFile.close();
  return 0;
}
```

  - `!scoreFile` returns true if the file is not created successfully.
- What will happen if we replace `scoreFile` by `cout`?

Input file streams

- To read data from a file, we create an input file stream.
- We create an **`ifstream`** object.

```
ifstream file object;
file object.open(file name);
// ...
file object.close();
```

```
ifstream myFile;
myFile.open("temp.txt");
// ...
myFile.close();
```

- The only open mode we will use for **`ifstream`** is **`iso::in`** (default).
- Again, we may use **`if(!myFile)`** to check whether a file is really opened.
  - If the file does not exist, **`myFile`** returns false.

Reading from an input file stream

- If the input data file is well-formatted, we may use the operator `>>`.
  - Like most of the testing input data for your Homework.
  - Those files that you may predict the type of the next piece of data.
- For example, suppose we have a file containing names and grades:
  - In each line, there is a name and one score (an integer).
  - Of course, they are separated by white spaces.
- How to calculate the average grades?
- How to find the one with the highest grades?
- How to generate a frequency distribution?

```
Tony 100
Alex 98
Robin 95
John 90
Mary 100
Bob 80
```

Reading from an input file stream

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

int main()
{
  ifstream inFile("score.txt");
  if(inFile)
  {
    string name;
    int score = 0;
    int sumScore = 0;
    int scoreCount = 0;
```

```
    while(inFile >> name >> score)
    {
      sumScore += score;
      scoreCount++;
    }
    if(scoreCount != 0)
      cout << static_cast<double>(sumScore)
              / scoreCount;
    else
      cout << "no grade!";
  }
  inFile.close();
  return 0;
}
```

```
Tony 100
Alex 98
Robin 95
John 90
Mary 100
Bob 80
```

- `>>` reads data **between** two spaces (or tabs or new line characters) and **tries to** convert that piece of data into the specified type.

End of file

- In each file, there is a special character “end of file”.
  - In C++, it is represented by the variable **`EOF`**.
  - It is always at the end of a file.
- When we do **`inFile >> name >> score`**:

```
Tony 100
Alex 98
```

T	o	n	y		1	0	0	\n	A	l	e	x		9	8	EOF
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

End of file

- In each file, there is a special character “end of file”.
  - In C++, it is represented by the variable **`EOF`**.
  - It is always at the end of a file.
- When we do **`inFile >> name >> score`**:

```
Tony 100
Alex 98
```

T	o	n	y		1	0	0	\n	A	l	e	x		9	8	EOF
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

End of file

- In each file, there is a special character "end of file".
  - In C++, it is represented by the variable **`EOF`**.
  - It is always at the end of a file.
- When we do **`inFile >> name >> score`**:

```
Tony 100
Alex 98
```

T	o	n	y		1	0	0	\n	A	l	e	x		9	8	EOF
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

End of file

- In each file, there is a special character "end of file".
  - In C++, it is represented by the variable **`EOF`**.
  - It is always at the end of a file.
- When we do **`inFile >> name >> score`**:

```
Tony 100
Alex 98
```

T	o	n	y		1	0	0	\n	A	l	e	x		9	8	EOF
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

End of file

- In each file, there is a special character “end of file”.
  - In C++, it is represented by the variable **`EOF`**.
  - It is always at the end of a file.
- When we do **`inFile >> name >> score`**:

```
Tony 100
Alex 98
```

`T`	**`o`**	**`n`**	**`y`**		**`1`**	**`0`**	**`0`**	**`\n`**	**`A`**	**`l`**	**`e`**	**`x`**		**`9`**	**`8`**	**`EOF`**
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

End of file

- In each file, there is a special character “end of file”.
  - In C++, it is represented by the variable **`EOF`**.
  - It is always at the end of a file.
- When we do **`inFile >> name >> score`**:

```
Tony 100
Alex 98
```

T	o	n	y		1	0	0	\n	A	l	e	x		9	8	EOF
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- An input operation (e.g., **`inFile >> name`**) returns false if it reads **`EOF`**.

End of file

- To verify that the current position is at the white space after a **>>** operation:

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

int main()
{
  ifstream inFile("test.txt");
  string name;
  char c = 0;
```

```
  if(inFile)
  {
    inFile >> name;
    c = inFile.get();
    cout << c << "-"; //   -
    c = inFile.get();
    cout << c << "-"; // 1-
  }
  inFile.close();
  return 0;
}
```

Reading from an input file stream

- Let’s modify the **`while`** loop:
  - The member function **`eof()`** returns true if the window is at **`EOF`**.

```
while(!inFile.eof())
{
  inFile >> name;
  inFile >> score;
  sumScore += score;
  scoreCount++;
}
```

Unformatted input files

- Sometimes a data file is not perfectly formatted.
  - We cannot predict what the next type will be.
  - For example, when there are missing values.
- In this case, we read data as characters and then manually find the types.
  - This process is called **parsing**.
- Some member functions of the class **`ifstream`**:
  - **`get()`** reads one character and returns it.
  - **`getline()`** reads multiple characters into a character array.

```
Tony 100
Alex 98
Robin
John 90
Mary 100
Bob 80
```

get() and getline()

- Let’s use **get()**:

```
while(!inFile.eof())
{
  char c = inFile.get();
  cout << c;
}
```

- Let’s use **getline()**:

```
while(!inFile.eof())
{
  char name[20];
  inFile.getline(name, 20);
  cout << name << endl;
}
```

getline() in a smarter way

- Let’s use **getline()** with a **delimiter**:

```
char name[20];
inFile.getline(name, 20, ' ');
cout << name << endl;
```

- **getline()** stops when the delimiter is read.
  - It must be a character.
  - It will be read and **discarded**.

```
inFile.getline(n, 100, ' ');
c = inFile.get();
cout << c << "-"; // 1-
c = inFile.get();
cout << c << "-"; // 0-
```

T	o	n	y		1	0	0	\n	A	l	e	x		9	8	EOF
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

getline() in a smarter way

- Let’s use **getline()** with a **delimiter**:

```
char name[20];
inFile.getline(name, 20, ' ');
cout << name << endl;
```

- **getline()** stops when the delimiter is read.
  - It must be a character.
  - It will be read and **discarded**.

```
inFile.getline(n, 100, ' ');
c = inFile.get();
cout << c << "-"; // 1-
c = inFile.get();
cout << c << "-"; // 0-
```

T	o	n	y		1	0	0	\n	A	l	e	x		9	8	EOF
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

`getline()` for C++ strings

- **Determining the types** and preparing a **large enough buffer** are always issues.
  - **C++ strings** may help.
- In particular, we may use the global function **`getline()`** in **`<string>`**.
  - The delimiter is also read and discarded.

```
istream& getline(istream& is, string& str, char delim);
```

- As an example:

```
while(!inFile.eof())
{
  string name;
  getline(inFile, name, ' ');
  cout << name << endl;
}
```

Updating a file

- How to update “Alex” to “Alexander”?
  - The member function **`seekp()`** moves the window.
  - What should we do when we are at ‘**`A`**’?
- Updating a file typically requires **copy-and-paste**.
  - Because plain text files are **sequential-access** files.
- The easiest way may be to read from the file, do modifications, and then write to a completely new file!

```
Tony 100
Alex 98
Robin 95
John 90
Mary 100
Bob 80
```

Updating a file

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

int main()
{
  ifstream inFile("test.txt");
  ofstream outFile("test1.txt");
  string name;
  int score;
```

```
  if(inFile && outFile)
  {
    while(inFile >> name >> score)
    {
      if(name == "Alex")
        name = "Alexander";
      outFile << name << " "
              << score << endl;
    }
  }
  inFile.close();
  outFile.close();
  return 0;
}
```

```
Tony 100
Alex 98
Robin 95
John 90
Mary 100
Bob 80
```

`>>` vs. `getline()`

- The two operations are similar but different:
  - `>>` tries to convert the piece into the specified type; **`getline()`** simply store that piece as a C or C++ string.
  - `>>` stops at the first character not of that type; **`getline()`** stops at one character after the delimiter.
- Suppose that the text file now may contain the first name and last name of a student, separated by a white space.
  - We use a colon to separate a name and a score.
- How to write a program to calculate the sum of scores?

```
Tony Wang: 100
Alex Chao: 98
Robin Chen: 95
Lin: 90
Mary: 100
Bob Tsai: 80
```

>> vs. getline()

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

int main()
{
  ifstream inFile("score.txt");
  string name;
  int score = 0;
  int sumScore = 0;
```

```
  if(inFile)
  {
    while(!inFile.eof())
    {
      getline(inFile, name, ':');
      inFile >> score;
      sumScore += score;
    } // good!
    cout << sumScore << endl;
  }
  inFile.close();

  return 0;
}
```

```
Tony Wang: 100
Alex Chao: 98
Robin Chen: 95
Lin: 90
Mary: 100
Bob Tsai: 80
```

>> vs. getline()

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

int main()
{
  ifstream inFile("score.txt");
  string name;
  int score = 0;
  int sumScore = 0;
```

```
  if(inFile)
  {
    while(!inFile.eof())
    {
      getline(inFile, name);
      inFile >> score;
      sumScore += score;
    } // bad! Why?!?!
    cout << sumScore << endl;
  }
  inFile.close();

  return 0;
}
```

```
Tony Wang
100
Alex Chao
98
Robin Chen
95
Lin
90
Mary
100
Bob Tsai
80
```

`>>` vs. `getline()`

- `>>` stops at **the first character not of that type**.
- After the `inFile >> score` operation, the input cursor stops at the **newline character**.
- The next `getline(inFile, name)` operation reads only the newline character into `name`.
  - The cursor gets to '`A`' in the third line.
- The next `inFile >> score` operation then fails to convert "`Alex`" into an integer.
- To fix this problem, we need to manually **bypass the newline character**.
  - The member function `ignore()` ignores one character.

```
Tony Wang
100
Alex Chao
98
Robin Chen
95
Lin
90
Mary
100
Bob Tsai
80
```

>> vs. getline()

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

int main()
{
  ifstream inFile("score.txt");
  string name;
  int score = 0;
  int sumScore = 0;
```

```
  if(inFile)
  {
    while(!inFile.eof())
    {
      getline(inFile, name);
      inFile >> score;
      inFile.ignore();
      sumScore += score;
    } // good!
    cout << sumScore << endl;
  }
  inFile.close();

  return 0;
}
```

```
Tony Wang
100
Alex Chao
98
Robin Chen
95
Lin
90
Mary
100
Bob Tsai
80
```

An alternative way

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

int main()
{
  ifstream inFile("score.txt");
  string name;
  string scoreStr;
  int score = 0;
  int sumScore = 0;
```

```
  if(inFile)
  {
    while(!inFile.eof())
    {
      getline(inFile, name);
      getline(inFile, scoreStr);
      score = stoi(scoreStr);
      sumScore += score;
    } // good!
    cout << sumScore << endl;
  }
  inFile.close();

  return 0;
}
```

```
Tony Wang
100
Alex Chao
98
Robin Chen
95
Lin
90
Mary
100
Bob Tsai
80
```

Outline

- C++ Strings
- File I/O
- **Self-defined header files**

Libraries

- There are many C++ standard **libraries**.
  - **`<iostream>`**, **`<fstream>`**, **`<cmath>`**, **`<cctype>`**, **`<string>`**, etc.
- We may also want to define **our own libraries**.
  - Especially when we collaborate with others.
  - Typically, one implements classes or global functions for the others to use.
  - That function can be defined in a self-defined library.
- A library includes a **header file** (.h) and a **source file** (.cpp).
  - The header file contains declarations
  - The source file contains definitions.

Example

- Consider the following program with a single function **`myMax()`**:

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

int myMax(int [], int);
int main()
{
  int a[5] = {7, 2, 5, 8, 9};
  cout << myMax(a, 5);
  return 0;
}
```

```
int myMax(int a[], int len)
{
  int max = a[0];
  for(int i = 1; i < len; i++)
  {
    if(a[i] > max)
      max = a[i];
  }
  return max;
}
```

- Let’s define a constant **variable** for the array length in **a header file**.

Defining variables in a library

myMax.h

```
const int LEN = 5;
```

main.cpp

```
#include <iostream>
#include "myMax.h"
using namespace std;

int myMax(int [], int);
int main()
{
  int a[LEN] = {7, 2, 5, 8, 9};
  cout << myMax (a, LEN);
  return 0;
}
```

```
int myMax(int a[], int len)
{
  int max = a[0];
  for(int i = 1; i < len; i++)
  {
    if(a[i] > max)
      max = a[i];
  }
  return max;
}
```

Including a header file

- When your main program wants to include a self-defined header file, simply indicate its path and file name.
  - `#include "myMax.h"`
  - `#include "D:/test/myMax.h"`
  - `#include "lib/myMax.h"`
  - Using \ or **/** does not matter (on Windows).
- We still compile the main program as usual.
- Let's also define **functions** in our library!
  - Now we need a source file.

Defining functions in a library

myMax.h

```
const int LEN = 5;
int myMax(int [], int);
```

main.cpp

```
#include <iostream>
#include "myMax.h"
using namespace std;

int main()
{
  int a[LEN] = {7, 2, 5, 8, 9};
  cout << myMax(a, LEN);
  return 0;
}
```

myMax.cpp

```
int myMax(int a[], int len)
{
  int max = a[0];
  for(int i = 1; i < len; i++)
  {
    if(a[i] > max)
      max = a[i];
  }
  return max;
}
```

Including a header and a source file

- When your main program also wants to include a self-defined source file, the include statement needs not be changed.
  - `#include "myMax.h"`
- We add a source file myMax.cpp.
  - In the source file, we **implement** those functions declared in the header file.
  - The main file names of the header and source files can be different.
- The two source files (main.cpp and myMax.cpp) must be **compiled together**.
  - Each environment has its own way.

Defining one more function

myMax.h

```
const int LEN = 5;
int myMax (int [], int);
void print(int);
```

main.cpp

```
#include <iostream>
#include "myMax.h"
using namespace std;

int main()
{
  int a[LEN] = {7, 2, 5, 8, 9};
  print(myMax(a, LEN));
  return 0;
}
```

myMax.cpp

```
int myMax(int a[], int len)
{
  int max = a[0];
  for(int i = 1; i < len; i++)
  {
    if(a[i] > max)
      max = a[i];
  }
  return max;
}
void print(int i)
{
  cout << i; // cout undefined!
}
```

Defining one more function

- Each source file contains statements to run.
- Each source file must include the libraries it needs for its statements.

```cpp
#include <iostream>
using namespace std;
int myMax(int a[], int len)
{
  int max = a[0];
  for(int i = 1; i < len; i++)
  {
    if(a[i] > max)
      max = a[i];
  }
  return max;
}
void print(int i)
{
  cout << i; // good!
}
```

The complete set of files

myMax.h

```
const int LEN = 5;
int myMax(int [], int);
void print(int);
```

main.cpp

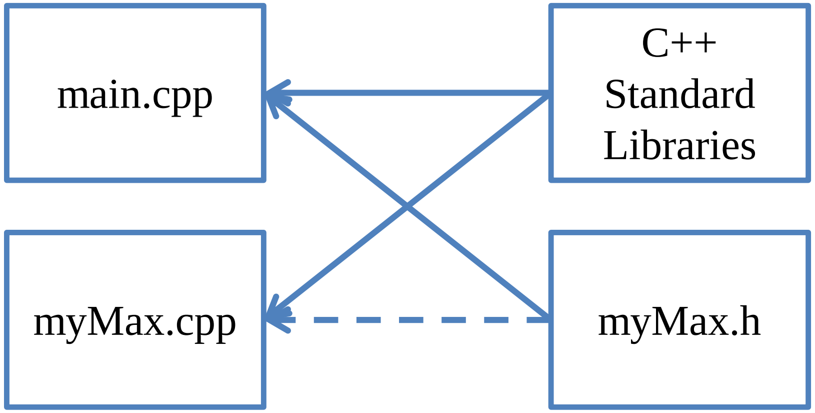
```
#include <iostream>
#include "myMax.h"
using namespace std;

int main()
{
  int a[LEN] = {7, 2, 5, 8, 9};
  print(myMax (a, LEN));
  return 0;
}
```

myMax.cpp

```
#include <iostream>
using namespace std;
int myMax(int a[], int len)
{
  int max = a[0];
  for(int i = 1; i < len; i++)
  {
    if(a[i] > max)
      max = a[i];
  }
  return max;
}
void print(int i)
{
  cout << i;
}
```

Remarks

- In many cases, myMax.cpp also include myMax.h.
  - E.g., if **`LEN`** is accessed in myMax.cpp.
- More will be discussed in further courses (e.g., Data Structures).
  - More than two source files.
  - A header file including another header file.


main.cpp
C++ Standard Libraries
myMax.cpp
myMax.h