Programming Design

C/C++ Strings and File I/O

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Applications of classes

- We have studied a lot about classes.
  - Encapsulation.
  - Constructors, copy constructors, destructors.
  - Operator overloading.
- Remaining topics:
  - Inheritance.
  - Polymorphism.
- Today let’s study two applications of classes.
  - C++ strings (and C strings).
  - File input/output.

Outline

- **C strings: character arrays**
- C++ Strings
- File I/O
  - Writing data to a file
  - Reading data from a file

Strings

- In many applications, we need some ways to handle **strings**.
- E.g., in an address book application, if we do not have strings:
  - We cannot store names.
  - We cannot store phone numbers.
  - We cannot store addresses.
- Strings can be implemented in two ways:
  - C strings as **character arrays**.
  - C++ strings as **objects**.
- Let’s introduce C strings first.

C strings as character arrays

- A C string is a character array.
- We have already used a string with **`cout`**:

```
cout << "Hello world";
```

  - **`"Hello world"`** is a string.
- A string is contained in a pair of double quotation marks.
  - A character is contained in a pair of single quotation marks.

C strings v.s. other arrays

- C strings are nothing but a character arrays.
- However, character arrays are “special”.
- For example:

```
int array[10];
cin >> array;
return 0;
```

```
char array[10];
cin >> array;
return 0;
```

  – While the first one results in a compilation error, the second one can run!

C strings v.s. other arrays

- For an array **`A`**, if we do `cin >> A`:
  - If **`A`** is of other types, this is not allowed.
  - But for a character array, this allows us to input the string.

```
char str[10];
cin >> str; // if we type "abcde"
cout << str[0]; // 'a'
cout << str[2]; // 'c’
```

C strings vs. other arrays

- For an array **A**, if we do **cout << A**:
  - If **A** is of other types, this will print out it memory address.
  - But for a character array, this prints out the whole string (some exceptions will be discussed later).

```
int values[5] = {0};
cout << values; // an address
char array[10] = {'a', 'b', 'c'};
cout << array; // "abc"
```

Input/output of a C string

- Because it is too often for a program to input/output a string, the C++ standard **implementents** **<<** and **>>** for character arrays in a **special** way.
  - **<<** and **>>** are operators.
  - An operator can do different things according to the input data types.
  - This is **operator overloading**!
- The implementation of C string I/O needs to be investigated in more details.
- Before that, let’s see how to declare a C string.

C string declaration and initialization

- A C string is declared as a character array.
  - `char s[100];`
- A C string may be **initialized** with a double quotation.
  - `char s[100] = "abc";`
  - Operator overloading again.
- In this case, a **null character** `\0` is appended at the end **automatically**.
  - `\0` is an escape sequence. It marks the **end of a string**.
  - The null character is `\0`, not `\o` or `\O`.
  - The length of the string stored in `s` is 3 + 1 (`\0`).
- When you declare a character array of length *n*, you can store a string of length at most ***n* – 1**.

Understanding the null character

- From the system's perspective, a null character marks the end of a string.
  - In particular, `<<` is implemented to print out characters up to `\0`.

```
char a[100] = "abcde FGH";
cout << a << endl; // abcde FGH
char b[100] = "abcde\0 FGH";
cout << b << endl; // abcde
```

- One may also initialize a C string by assigning multiple characters.
  - `char s[100] = {'a', 'b', 'c'};`
  - **No** null character will be appended.
  - `=` is overloaded for "a C string" and "some characters" in different ways.

String assignments

- Assignments with double quotations are allowed only for initialization.
  - `char s[100];`
    `s = "this is a string"; // compilation error!`
- One may assign values to a string by assigning characters.
  - `s[0] = 'A'; s[1] = 'B'; s[2] = 'C';`
- One may assign values by `cin >>`.
  - `cin >> s;`
  - A null character will be appended.

```
char c[100];
cin >> c; // "123456789"
cin >> c; // "abcde";
cout << c << endl; // "abcde"
c[5] = '*';
cout << c << endl; // "abcde*789"
```

Array boundary

```
char a[5];
cin >> a; // "123456789"
cout << a; // "123456789" or an error
```

- C++ does not check **array boundary**!
- We may or may not touch those memory spaces used by other programs/variables.
  - If a protected space is touched, an error occurs and our program is shutdown.
  - If not, `cout <<` prints out **the whole string** until the **end of a string**, which is marked by a `\0`.

A strange case

```
char a1[100];
cin >> a1; // "this is a string"
cout << a1; // "this"
```

- Is it because that a white space is treated as an end of C strings?
- No!

```
char a2[100] = {'a', 'b', ' ', 'c', '\0', 'e'};
cout << a2; // ab c
```

- Then why?

`cin >>` vs. `cin.getline()`

- When **`cin >>`** reads a white space, it treats that as the end of input and thus only “this” is stored into the array.
  - The same thing happens for a new line or a tab.
- To input a string with white spaces, use `cin.getline()`.
  - A instance function of the object **`cin`**.
  - It treats only end of line as the end of input.

```
char a[100];
cin.getline(a, 100); // "this is a string"
cout << a << endl; // "this is a string"
```

Useful functions for C strings

- Look at your textbook or websites to find some useful function.
- In **`<cstring>`**:
  - **`strlen()`**, **`strcat()`**, **`strcmp()`**, **`strchr()`**, **`strstr()`**, etc.
- In **`<cstdlib>`**:
  - **`atoi()`**, **`atof()`**, etc.
- For more powerful functionalities, let’s use C++ strings.

Outline

- C strings: character arrays
- **C++ Strings**
- File I/O
  - Writing data to a file
  - Reading data from a file

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>`.
- The C++ string is more convenient and powerful than C string.
- To use C++ strings, `#include <string>`.
- In the class `string`, there are:
  - A **member variable**, which is a character array whose length can vary.
  - Many **member functions**.
  - Many **overloaded operators**.

`string` declaration

- **`string myString;`**
- **`string myString = "my string";`**
  - **`string`** is a class defined in **`<string>`**.
  - **`string`** is not a C++ keyword.
  - **`myString`** is an object.
- A C++ string does not need a null character.
- We may use the member function **`length()`** to get the number of characters.
  - e.g., **`myString.length()`** returns 9.

string assignment

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

```
string myString = "my string";
string yourString = myString;
string herString;
herString = yourString;
herString = "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 **+**.

```
string myString = "my string ";
string yourString = myString;
string herString;
herString = myString + yourString;
  // "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 `[]`.
- Thanks to operator overloading!

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

`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.
- Use `getline(cin, a string object)`.
  - This is a global function defined in `<string>`.

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

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

Substring

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

```
substr(begin index, # of characters)
```

- As an example:

```
string s = "abcdef";
string b = s.substr(2, 3);
  // b == "cde"
```

string finding

- We may use the member function **find()** to look for a string or character.
- This will return the beginning index of the argument, if it exists, or **string::npos**, which is an integer defined in the namespace **string**, if not found.
- String literals or C strings can also be the argument.

```
find(a string)
```

```
string s = "abcdefg";
int i = s.find("bcd"); // i == 1;
string t;
cin >> t;
if(t.find("a") == string::npos)
  cout << "not containing a";
```

`string` comparison and modification

- We may use **`>`**, **`>=`**, **`<`**, **`<=`**, **`==`**, **`!=`** to compare two C++ strings.
- It is easy to find the comparison rule by yourself.
- 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.
- We may use **`insert()`**, **`replace()`**, and **`erase()`** to modify a string.
- Look up these functions of string, and more, from books or websites.

`string` for unformatted input files

- For an unformatted input file, we used `getline()` or `>>` with C strings.
  - The length of our buffer is always an issue.

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

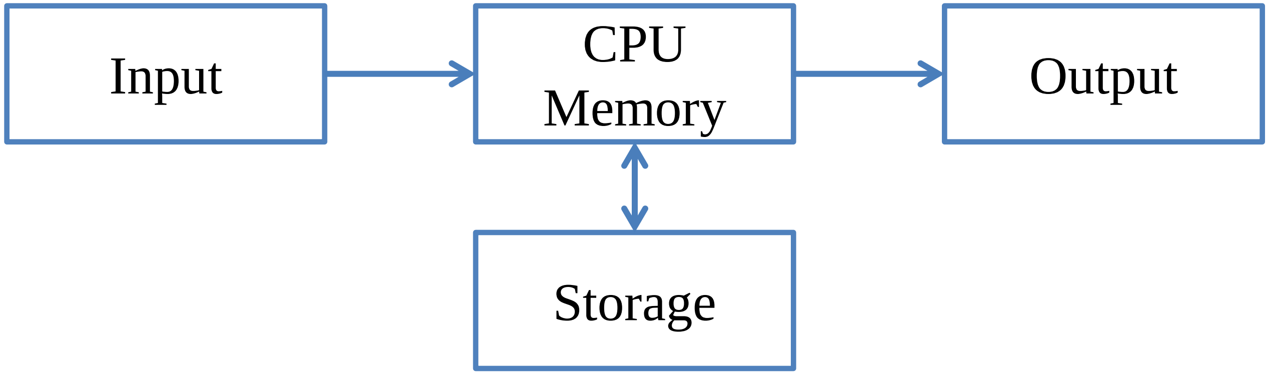
- We may use C++ string instead!

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

Outline

- C strings: character arrays
- C++ Strings
- **File I/O**
  - **Writing data to a file**
  - Reading data from a file

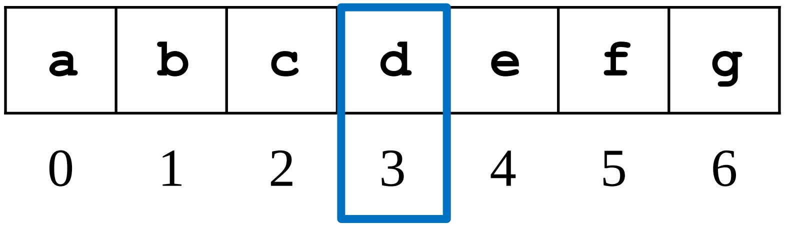
File I/O



- 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

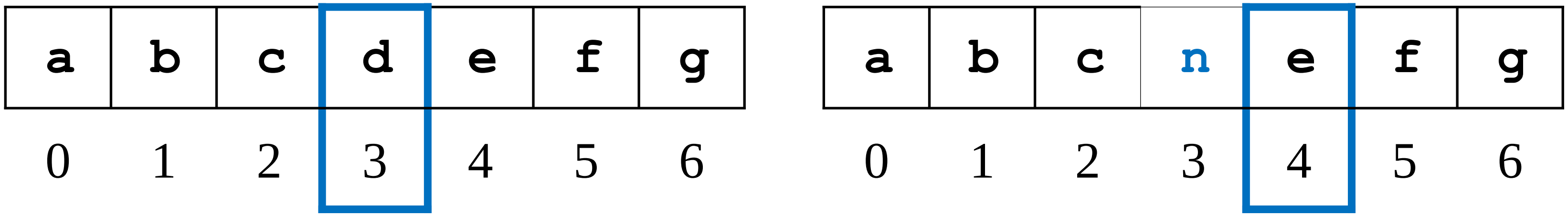
- Files store data.
  - A plain-text file stores **characters**.
  - A MS Word document stores characters and **format** information.
  - A bitmap file stores **color** codes.
- How are characters stored in a plain-text files?
  - Each character has its own **position**.
  - For each opened file, there 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 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* is a C string.
- Do you care about the following questions?
  – Is there a member variables storing the file name?
  – How are `open()` and `close()` 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`**?

Outline

- C strings: character arrays
- C++ Strings
- **File I/O**
  - Writing data to a file
  - **Reading data from a file**

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
Adam 98
Robin 95
John 90
Mary 100
Bob 80
```

Reading from an input file stream

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

int main()
{
  ifstream inFile("score.txt");

  if(inFile)
  {
    char name[20] = {0};
    int score = 0;
    int sumScore = 0;
    int scoreCount = 0;
```

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

  return 0;
}
```

```
Tony 100
Adam 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
Adam 98
```

`T`	`o`	`n`	`y`		`1`	`0`	`0`	`\n`	`A`	`d`	`a`	`m`		`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`**.

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:
  - **`get()`** reads one character and returns it.
  - **`getline()`** reads multiple characters into a character array.

```
Tony 100
Adam 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())
{
  inFile.getline(name, 20);
  cout << name << endl;
}
```

`getline()` in a smarter way

- Let’s use **`getline()`** with **the third argument**:

```
while(!inFile.eof())
{
  inFile.getline(name, 20, ' '); // inFile >> name;
  cout << name << endl;
}
```

- **`getline()`** stops when the third argument is read.
  - The third argument must be a character.
- **Determining the types** and preparing a **large enough buffer** are always issues.
  - **C++ strings** will help us.

Updating a file

- How to update "Adam" 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.
- How to read from or write to **random-access** files?

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