# Programming Design

# C++ Strings

Ling-Chieh Kung

Department of Information Management
National Taiwan University

# Applications of classes

- Let’s study an applications of classes.
  - C++ strings.

# 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;
}
```