IM 1003: Computer Programming
Strings

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Strings

- In many applications, we need some ways to handle **strings**.
- E.g., in the address book homework, 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**.

Outline

- **C strings**
- C++ strings

Strings

- A C string is a character array.
- We have already used string with **`cout`**:
  - **`cout << "Hello world";`**
- **`"Hello world"`** is a string.
- A string is contained in a pair of double quotation.
  - A character is contained in a pair of single quotation.

C strings v.s. other arrays

- C strings are nothing but a character array created by the programmer.
- However, they are “special”.
- For example:

```
int main()
{
  int array[10];
  cin >> array;
  return 0;
}
```

```
int main()
{
  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 C string (character array), this allows us to input the string.

```
int main()
{
  char array[10];
  cin >> array; // if we type "abcde"
  cout << array[0]; // 'a'
  cout << array[2]; // 'c'
  return 0;
}
```


C strings v.s. 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 C string (character array), this prints out the whole string (some exceptions will be discussed later).

```
int main()
{
  char array[10];
  array[0] = 'a';
  array[1] = 'b';
  array[2] = 'c';
  cout << array; // "abc"
  return 0;
}
```


Input/output a C string

- Because it is too often for a program to input/output a string, the C++ standard **implements** **<<** and **>>** for character arrays in a **special** way.
  - **<<** and **>>** are operators.
  - This scheme is called **operator overloading**, which will be discussed later.
- The implementation of C string input/output needs to be investigated in more details.
- Before that, let’s see how to declare a C string.

String declaration and initialization

- A string is declared as a character array.
  - `char t[100];`
- A string may be initialized by using a double quotation.
  - `char s[100] = "this is a string";`
- Whenever we initialize a string with double quotations, a **null character** `\0` is appended at the end **automatically**.
  - This marks the **end of a string**.
  - Therefore, length of the string stored in **s** is 13 + 3 (spaces) + 1 (`\0`).

String declaration and initialization

- 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 multiple characters.
  - `s[0] = 't'; s[1] = 'h'; s[2] = 'i'; // and so on`
  - **No** null character will be appended. We need to do this by ourselves.
- Alternatively, one may assign values by `cin >>`.
  - `cin >> s;`
  - **A** null character will also be appended.
  - Suppose I enter "yeah!", the length is 5 + 1.

Understanding the null character

- The null character is one of the escape sequences.
  - It is `\0`, not `\o` or `\O`.
- A null character marks the end of a string.
- It may be added into a string by the programmer.

```
char a[100] = "abcde FGH";
cout << a; // abcde FGH
```

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


Input/Output a C string

- Consider the following three scenarios:

```
char a[100];
cin >> a; // input "Hello!"
cout << a; // why not all the 100 characters?
```

```
char a[5];
cin >> a; // Try to input "1234567890"
cout << a; // Why not only "12345"? Why error?
```

```
char a[100];
cin >> a; // Try to input "this is a string"
cout << a; // Why "this" only?
```

Input/Output a C string

```
char a[100];
cin >> a; // input "Hello!"
cout << a; // why not all the 100 characters?
```

- When one uses **cin**, C++ **always appends** a **\0** after the input string. This **\0** is a mark of "**end of string**".
  - **\0** is the "null character".
- So the "string" is printed out, not the whole array.

Input/Output a C string

```
char a[5];
cin >> a; // Try to input "1234567890"
cout << a; // Why not only "12345"? Why 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 <<** is implemented to print out **the whole string** until the **end of a string**, which is marked by a **\0**.

Input/Output a C string

```
char a[100];
cin >> a; // Try to input "this is a string"
cout << a; // Why "this" only?
```

- When C++ outputs a string, it always treats the first null character as the end of string.
- Nevertheless, a white space is **not** a null character!

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

- Then why "this" only?

Understanding the null character

- It is because when **cin >>** read a new line, a white space, or a tab, it will treat it as the end of input, thus the only "this" is stored into the array.
- To solve this situation, use **cin.getline();**.
  - **cin** is an object defined in **<iostream>**.
  - **getline()** is a member function defined in the class of **cin**.
  - **cin.getline()** treat only end of line as the end of input.

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

Understanding the null character

- A C string ends with a null character **\0**.
- Since an initialization with double quotations and **cin >>** append null characters for us, usually you do not need to worry about this.
  - When something goes wrong, check it.
- At least remember one thing:
  - When you declare a character array of length *n*, you can store a string of length at most *n* – 1.

A strange problem

- Consider the following example:

```
char a[100];
int b;
cin >> b;
cin.getline(a, 100);
cout << a;
```

- When we input a number and then press "enter":
  - 123 is received by **b**, **\n** is received by **a**.
  - Because **\n** is not a part of a number but it can be part of an array, it will be treated as the only input for **getline()**.

Fix the Strange Problem

- To fix this problem:

```
char a[100];
int b;
cin >> b;
cin.get(); // receives \n
cin.getline(a, 100);
cout << a;
```

- You may always add **cin.get()** after every **cin >>** for a numeric variable.
  - **cin.get()**: input a single character.
  - You have to append a **\0** by yourself.
- No language is perfect.

Useful functions

- Look at your book or the website to find those useful function.
- Mostly in **<cstring>**.
- **atoi()** (array to integer) and **itoa()** (integer to array) are in **<cstdlib>**.

Pointers and character arrays

- We have already known that a character array is a C string.
- As we may use a character pointer to represent a character array, we may use a character pointer to represent a string.

Pointers and character arrays

- For example:

```
char* x = "abcde";
cout << x << endl; // abcde
char* y = x + 2;
cout << y << endl; // cde
```

  - **x** and **y** are indeed pointers. But when we try to print them out, a string instead of an address is printed out.
  - This is because character pointers are treated as character arrays.
  - Six bytes will be allocated to store **abcde** and a null character.
  - Thus we can save some memory spaces by using **char***.

Pointers and character arrays

- However, we can only use **char*** when we also do initialization.
- If not, we will use a pointer variable without memory space allocated to it.

```
char* x;
cin >> x; // run time error
```


C string arrays

- It looks like a two-dimensional array (actually it is).
- Each row represent a string.

Outline

- C strings
- **C++ strings**

C++ Strings: `string`

- From now on, we'll say:
  - 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. We'll learn to use it right now.
- 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**.

`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.
- You can use the member function **`length()`** to get the number of characters in a string.
  - 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;
```

string concatenation

- 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.

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

- **+=** also works.

string indexing and input

- When we want to access a character in a C++ string, we may treat it as a usual array.

```
string myString = "my string";
char a = myString[5]; // r
```

- When we use **cin >>** to input into a C++ string, white spaces may still create problems.
- To fix this, now we cannot use **cin.getline()**.
  - The first argument of **cin.getline()** can only be a character array.
  - We will use another function instead.

string input: getline()

- Use **getline(cin, *a string object*)**.
  - This is 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 comparison

- 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.
  - However, if none of the two sides is a C++ string, there will be an error.

string finding

- We may use the member function **find()** to look for a string or character in a string.

```
find(a string)
```

- This will return the beginning index of the argument, if it exists, or **string::npos**, which is a variable in the namespace **string**, if not found.
- String literals or C strings can also be the argument.

string finding

- As an example:

```
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 modifying

- We may use **insert()**, **replace()**, and **erase()** to modify a string.
- Look up these functions of string, and more, from your book or a website.