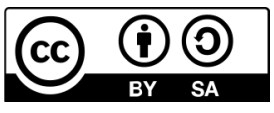
@prog2ua

Unit 2: The `string` class

Programming 2

Degree in Computer Engineering
University of Alicante
2023-2024



Index

1. Character arrays in C
2. The `string` class in C++
3. Type conversions
4. Comparison
5. Exercises

Character arrays in C

Declaration (1/3)

- *Character arrays* contain a sequence of `char` elements ending with the null character ('`\0`'):

```
// The compiler automatically puts the '\0' at the end
char str[]="hello";
// Another way of initialising, character by character
char str[]={'h','e','l','l','o','\0'};
// Missing '\0': not a valid character array
char str[]={'h','o','l','a'};
```

- Many functions that work with character arrays* look for the '`\0`' to identify where the array ends
- If there is no '`\0`' in the array, the result of these functions may not be as expected

*Such as those defined in the `string.h` library, as described later

Declaration (2/3)

- Character arrays in C have a fixed size and cannot be resized after being declared:

```
char str[10]; // Stores a maximum of 10 elements
```

- A space must always be reserved to store the null character (`'\0'`):

```
char str[10]; // Stores a maximum of 9 characters and
    '\0'
```

- They can be initialised when declared. In that case, it is not necessary to set the size:

```
char str[]="hello"; // Size 6 (5 letters + '\0')
char str2[10]="hello"; // Size 10, but only 6 are used
```

- Character arrays in C can also be used in C++

Declaration (3/3)

- Common errors when declaring character arrays:

```
// Array too small to store the string
char str[5]="parallelepiped"; // Compilation error

// Single quotes (') used instead of double quotes (")
char str[]='h'; // Compilation error
char str[]='hello'; // Compilation error

// Size not set and variable not initialised
char str[]; // Compilation error

// Attempt to assign a value with '=' after declaration
char str[10];
str="hello"; // Compilation error
```

Screen output

- Screen output with `cout` and `cerr` as with any of the other basic data types (`int`, `float`, etc.)
- Otput can combine variables, constants and different data types:

```
char str[]="Mark";
int num=10;

cout << str << " -> " << num; // Output is "Mark -> 10"
```

Keyboard input > Operator >> (1/2)

- Character arrays can be read from the keyboard, as in other basic data types, using `cin` and the operator `>>`
- There are some differences when reading from the keyboard with respect to other data types
- Blanks* before the string are ignored:

```
char str[32];
cin >> str;
// User writes "   hello"
// The str variable stores "hello"
```

*We mean with ”blank” a space, tab or new line (`'\n'`)

Keyboard input > Operator >> (2/2)

- Reading finishes as soon as the first white is found. Therefore, an entire string containing blanks cannot be read:

```
char str[32];
cin >> str;
// The user writes "good afternoon"
// The str variable stores "good"
```

- There is no limit in the number of characters that are read. User can type a string larger than the array size:

```
char str[5];
cin >> str;
// The user writes "sternocleidomastoid"
// Could overlap memory cells not belonging to the
    variable and produce a segmentation fault
```

Keyboard input > `getline` (1/4)

- Keyboard input can be also read using `cin` and the `getline` function
- This function allows reading strings with blanks, limiting the number of characters to be read:

```
const int SIZE=100;
char str[SIZE];
// str: variable where the characters are stored
// SIZE: number of characters read
cin.getline(str,SIZE);
// If the user enters "good evening"
// the variable str stores "good evening"
```

- Reads a maximum of `SIZE-1` characters or until reaching the end of the line
- The `'\n'` at the end of the line is read but not stored in the variable
- The function adds `'\0'` to the end of what has been read (therefore only reads `SIZE-1` characters)

Keyboard input > `getline` (2/4)

- If the user types more characters than indicated, they remain in the keyboard *buffer* and the next reading fails:

```
char str[10];
cout << "String 1: ";
cin.getline(str,10);
cout << "Read 1: " << str << endl;
cout << "String 2: ";
cin.getline(str,10);
cout << "Read 2: " << str << endl;
```

Terminal

```
$ myProgram
String 1: hello everybody
Read 1: hello eve
String 2: Read 2:
```

Keyboard input > `getline` (3/4)

- There can be problems when reading from `cin` combining the `>>` operator and the `getline` function:

```
int num;
char str[100];

cout << "Num: ";
cin >> num;
cout << "Input string: " ;
cin.getline(str,100);
cout << "What I read is: " << str << endl;
```

Terminal

```
$ myProgram
Num: 10
Input string: What I read is:
```

Keyboard input > `getline` (4/4)

- Why is this happening?
  - The `>>` operator reads `10`, but stops reading when the first non-numeric character is found (`'\n'` in this case)
  - The first thing that `getline` finds in the *buffer* is a `'\n'`, so it finishes reading and does not store anything in `str`
- Solution:

```
...
cin >> num;
cin.get(); // Add this line
           // Gets '\n' out of buffer
// getline can now be used without issues
...
```

The `string.h` library (1/2)

- The `string.h` library contains a set of functions that facilitate working with character arrays
- The library must be included in the code using it:

```
#include <string.h>
```

- `strlen` returns the length (number of characters) of a character array:

```
char str[10]="hello";
cout << strlen(str); // Prints 5
```

- `strcpy` copies one character array into another. Be careful not to exceed the size of the target array:

```
char str[5];
strcpy(str,"cool"); // The string fits into str: 4 + '\0'
     = 5 characters
strcpy(str,"house"); // No fitting! Segmentation fault!
```

The `string.h` library (2/2)

- `strcmp` compares two strings in lexicographical order*, returning `1` if `str1>str2`, `0` if `str1==str2` and `-1` if `str1<str2`:

```
char str1[]="root";
char str2[]="river";
cout << strcmp(str1,str2) << endl; // Prints 1
cout << strcmp(str2,str1) << endl; // Prints -1
cout << strcmp(str1,str1) << endl; // Prints 0
```

- The `strncmp` y `strncpy` functions compare or copy only the first `n` characters:

```
char str[8];
strncpy(str,"hello, world",5); // Copies only "hello"
str[5]='\0';  // Does not add '\0' automatically
              // We have to add it manually
```

*Order followed by words in a dictionary

Conversion to `int` and `float`

- To transform a character array to `int` or `float` the functions `atoi` or `atof` can be used
- These functions are defined in the library `cstdlib`:

```
#include <cstdlib> // Required when atoi/atof are used

char str[]="100";
int num=atoi(str); // num is 100

char str2[]="10.5";
float num2=atof(str2); // num2 is 10.5
```

The `string` class in C++

Definition (1/2)

- Character arrays in C can be used in C++, but C++ also has the `string` class* that allows working more easily and flexibly with character strings:

```
// Declaration of a string variable
string s; // No need to set the string size
// Declaration with initialisation
string s2="Alicante";
// Declaration of a constant
const string GREET="hello";
```

*More information on what a “class” is in Unit 5

Definition (2/2)

- A `string` has a variable size and can dynamically grow depending on the storage needs of the program:

```
string s="hello"; // Stores 5 characters
s="hello everybody"; // Stores 15 characters*
s="ok"; // Stores 2 characters
```

- No need to worry about the `'\0'`
- The passing of parameters (value and reference) is done as with any basic data type:

```
void myFunction(string s1,string &s2){
  // s1 is passed by value
  // s2 is passed by reference
}
```

*A white space counts as any other character

Screen output

- Screen output with `cout` and `cerr` as with character arrays in C:

```
string s="Mark";
int num=10;

cout << s << " -> " << num; // Prints "Mark -> 10"
```

Keyboard input > Operator >>

- `cin` and the `>>` operator can be used to read from keyboard in the same way as with character arrays in C
- Blanks before the string are ignored and reading finishes when the first blank is found:

```
string s;
cin >> s;
// User writes "    hello"
// The s variable stores "hello"
...
// User writes "good afternoon"
// The s variable stores "good"
```

Keyboard input > `getline` (1/2)

- As with character arays, the function `getline` can be used to read `string` variables
- Reading strings containing blanks is possible in this case:

```
string s;
getline(cin,s);
// If the user writes "good afternoon"
// the variable s stores "good afternoon"
```

- Does not limit the characters read, because with the `string` class is not necessary
- Alert! The syntax changes with respect to character arrays in C

Keyboard input > `getline` (2/2)

- If the `>>` operator and `getline` are combined while reading, there is the same problem as with character arrays in C*
- By default, `getline` reads until it finds the newline character (`'\n'`)
- An additional parameter can be passed to indicate that the function must read up to a specific character:

```
string s;
// Reads until finding the first comma
getline(cin,s,',');
// Reads until finding the first square bracket
getline(cin,s,'[');
```

*The solution is the same proposed in slide 11

Extracting words from a `string`

- Words can be easily extracted from a `string` by using the `stringstream` class:

```
#include <sstream> // Required when using stringstream
...
stringstream ss("Hello cruel world 666");
string s;

// Each iteration of the loop reads until reaching a
    blank
while(ss>>s){ // Extracts words one by one
  cout << "Word: " << s << endl;
}
```

`string` methods (1/3)

- Since `string` is a class, methods are called by putting a dot after the name of the variable
- `length` returns the number of characters in the string:

```
// unsigned int length()
string s="hello, world";
cout << s.length(); // Prints 12
```

- `find` returns the position in which a substring appears within a string:

```
// size_t find(const string &s,unsigned int pos=0)
cout << s.find("world"); // Prints 7
// If the substring is not found returns string::npos
```

`string` methods (2/3)

- `replace` substitutes a string (or part of it) with another one:

```
// string& replace(unsigned int pos,unsigned int len,
    const string &s)
string s="hello world";
s.replace(0,5,"hola"); // s is "hola world"
```

- `erase` allows removing part of a string:

```
// string& erase(unsigned int pos=0,unsigned int len=
    string::npos);
string s="hello world";
s.erase(4,3); // s is "hellorld"
```

- `substr` returns a substring of the original string:

```
// string substr(unsigned int pos=0,unsigned int len=
    string::npos) const;
string s="hello world";
string subs=s.substr(2,5); // subs is "llo w"
```

`string` methods (3/3)

- Example of use:

```
string a="There is a mug in this kitchen with mugs";
string b="mug";
unsigned int size=a.length(); // Length of a
// Search for the first "mug"
size_t found=a.find(b);
if(found!=string::npos){
  cout << "First in: " << found << endl;
  // Search for the second "mug"
  found=a.find(b,found+b.length());
  if(found!=string::npos)
    cout << "Second in: " << found << endl;
}
else{
  cout << "Word '" << b << "' not found";
}
// Replace the first "mug" with "bottle"
a.replace(a.find(b),b.length(),"bottle");
cout << a << endl;
```

Operators (1/2)

- Comparisons: == (equal), != (different), > (greater), >= (greater or equal), < (less) and <= (less or equal)

```
string s1,s2;
cin >> s1; cin >> s2;
if(s1==s2) // Comparison in lexicographical order
  cout << "Equal" << endl;
```

- Assignment of one string to another with the operator =, like any basic data type:

```
string s1="hello";
string s2;
s2=s1;
```

- String concatenation with the operator +:

```
string s1="hello";
string s2="world";
string s3=s1+","+s2; // s3 is "hello, world"
```

Operators (2/2)

- Access to components with the operator `[]`, as with character arrays in C:

```
string s="hello";
char c=s[4]; // s[4] is 'o'
s[0] = 'H';
cout << s << ":" << c << endl ; // Prints "Hello:o"
```

- Characters cannot be assigned to positions outside the `string`:

```
string s;
s[0]='g'; s[1]='o'; s[2]='o'; s[3]='d';
// Does not store anything because s is an empty string
    and these positions are not reserved
```

- Example of traversal of a `string` character by character:

```
string s="hello, world";
for(unsigned int i=0;i<s.length(); i++)
  s[i]='f'; // Replaces each character with 'f'
```

Type conversions

Conversion between `string` and character array in C

- A character array in C can be assigned to a `string` using the assignment operator (=):

```
char str[]="hello";
string s;
s=str;
```

- A `string` can be assigned to a character array in C using `strcpy` and `c_str`:*

```
char str[10];
string s="world";
// There must be enough room in str
strcpy(str,s.c_str());
```

*The `c_str` method returns a character array in C with the contents of the `string`

Conversion between `string` and number

- Transform an integer or real number to `string`:

```
#include <string> // It is not the same as <string.h>
...
int num=100;
string s=to_string(num);
```

- Transform a `string` to integer:*

```
string s="100";
int num=stoi(s);
```

- Transform a `string` to real number:

```
string s="10.5";
float num=stof(s);
```

*The functions `to_string`, `stoi` and `stof` are available from C++ 2011 version onward

Comparison

Character array in C vs. `string`

Character array in C	`string`
`char str[SIZE];`	`string s;`
`char str[]="hello";`	`string s="hello";`
`strlen(str)`	`s.length()`
`cin.getline(str,SIZE);`	`getline(cin,s);`
`if(!strcmp(str1,str2)){...}`	`if(s1==s2){...}`
`strcpy(str1,str2);`	`s1=s2;`
`strcat(str1,str2);`	`s1=s1+s2;`
`strcpy(str,s.c_str());`	`s=str;`
Ends with `'\0'`	Does not end with `'\0'`
Fixed allocated size	Variable allocated size
Variable used size	Used size == allocated size
Can be used with binary files	Cannot be used with binary files

Exercises

Exercises (1/4)

Exercise 1

Code a function called `subString` that returns a substring of length `n`, starting at position `p` of other string. Both the argument and the return value must be `string` type.

```
subString("heeello",2,5) // Returns "lo"
```

Exercise 2

Code a function `deleteStringCharacter` that, given a `string` and a character, deletes all the occurrences of that character in the `string` and returns it.

```
deleteStringCharacter("cocobongo",'o') // Returns "ccbng"
```

Exercises (2/4)

Exercise 3

Code a function `searchSubstring` that searches the first occurrence of a substring `a` inside a string `b` and returns its position, or `-1` if not found. Both `a` and `b` must be `string` type.

```
searchSubstring("eel","heeello") // Returns 2
```

Extensions:

1. Add another parameter to the function that indicates the number of occurrence to return (if the value is `1` it would work as the original function)
2. Implement another function that returns the number of occurrences of the substring in the string

Exercises (3/4)

Exercise 4

Code a function `encrypt` that encodes a string by adding a number `n` to the ASCII code of each character, taking into account that the result must be a character.

For example, if `n=3`, `a` is encoded as `d`, `b` as `e`,..., `x` as `a`, `y` as `b`, and `z` as `c`.

The function must admit lowercase and uppercase letters. Non-letter characters must not be encoded. The parameter must be `string` type.

```
encrypt("hello, world",3) // Returns "khoor, zruog"
```

Exercises (4/4)

Exercise 5

Write a function `isPalindrome` that returns `true` if the `string` parameter is a palindrome.

```
isPalindrome("racecar") // Returns true
isPalindrome("hello, olleh") // Returns false
```

Exercise 6

Implement a function `createPalindrome` that adds to a `string` the same `string` but reversed so that the result is a palindrome.

```
createPalindrome("hello") // Returns "helloolleh"
```