

# Strings and Streams

# Representing Text, Working with Streams from Files and Strings



## SoftUni Team

## Technical Trainers



# SoftUni



Software University

<https://softuni.bg>

sli.do

**#c++-fundamentals**

## 1. Text in Computers

## 2. Text Representation

- C-Strings
- The `std::string` class

## 3. Streams

- Streaming to files
- Streaming from files
- The `std::stringstream` class





# **Text in Computers**

Bytes, Code Points, Encoding

- Data is **bytes** of **1s** and **0s**
- Ways we interpret bytes:
  - **Binary number** -> integer types
  - **IEEE754** -> floating-point types
  - **Binary "code point"** -> char types
- **Characters** are just another interpretation of binary data

128	64	32	16	8	4	2	1
1	0	0	1	1	0	1	1
<hr/>							
$128 + 0 + 0 + 16 + 8 + 0 + 2 + 1$							
$= 155$							

# Text in Computers

- **Text is a sequence of characters**
- A character consists of **one or more bytes**
  - The binary representation of a **number**
  - Interpreted as a **code point** from a **character set**
- **Character set** – a group of characters
- **Code point** – unique number assigned to a character in a charset
  - ASCII code point **65** is '**A**'



- **ASCII** is the base charset – code points from **0** to **127**
  - English letters, digits, punctuation, control symbols
- **Extended ASCII** – code points from **128** to **255**
  - Different charsets use those codepoints for different characters
  - Windows Cyrillic code point **211** (**0xD3**) is **Ѹ**
  - Windows Greek code point **211** (**0xD3**) is **Σ**
- Unicode unifies charsets to represent **all the world's characters**



# **Text Representation**

## C-Strings and `std::string` class



- C++ has good native support for the **ASCII charset**
  - **char** data type covers code points 0 to 255
- **Text types** (sequences of characters) are called **strings**
- C++ has two standard ways of working with text
  - **Character arrays**
  - **std::string** – a "smart" wrapper of a C-String

- An array of char (**char str[]** or **char\* str**) with the following rules:
  - Should be null-terminated: end with '**\0**' which is **char(0)**
  - '**\0**' counts as an element – it affects array size
- Null-terminator tells C++ where the string ends
  - C++ arrays don't know their size

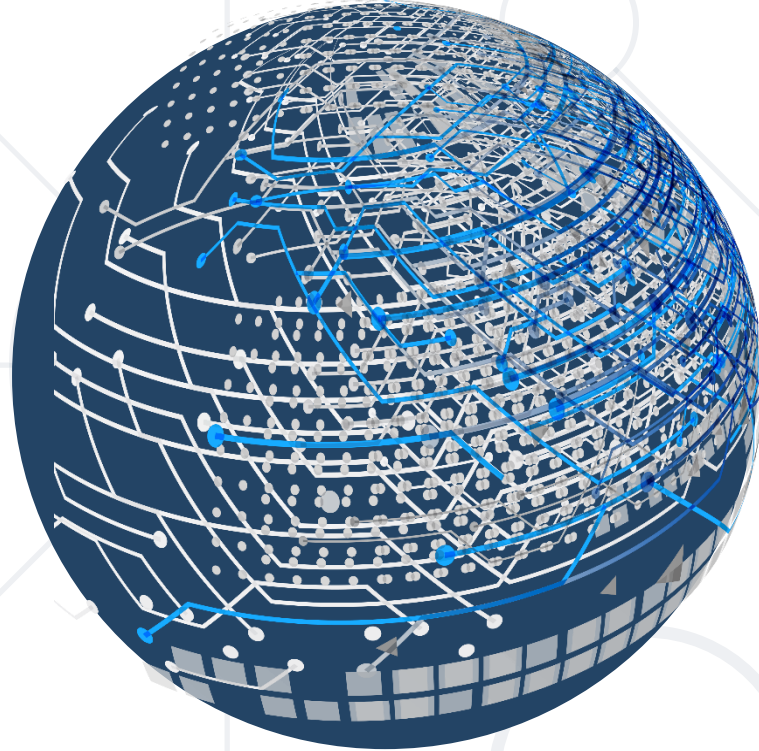


- Initialization can happen with **array initializer** or **literal**
  - If using normal array initializer, don't forget the '**\0**' at the end

```
char text[12] = { 'C', '+', '+', ' ', 'P', 'r', 'o', 'g', 'r', 'a', 'm', '\0' };  
char sameText[] = { 'C', '+', '+', ' ', 'P', 'r', 'o', 'g', 'r', 'a', 'm', 0 };  
char sameTextAgain[] = "C++ Programming";  
char sameTextYetAgain[12] = "C++ Programming";
```

- **cin** and **cout** can directly write to and read from C-Strings
  - **cout** prints until it reaches '**\0**'
  - **cin** works correctly only if array can fit entered data

- C-String functions are defined in the `<cstring>` header
- `strcat(char* destination, const char* source)`
  - Appends (concatenates) `source` C-String into `destination` C-String
  - `destination` needs to be long enough for source + null-terminator
- `strlen(const char* str)`
  - Returns length of C-String in `str`
- `strstr(const char* str, const char* search)`
  - Returns the address of `search` in `str` and `NULL` if not found
  - `int index = strstr(str, search) - str;` gets the index



**std::string Class**

- The C++ **string** encapsulates a null-terminated C-String
  - **#include<string>**
- Declare like a normal variable
  - Empty (**"**, size **0**) if only declared
  - Can be initialized with C-String or string literal

```
string theFoxPart = "the quick brown fox";  
string theActionPart("jumps over");  
char dogPartCString[] = "the lazy dog";  
string sentence = theFoxPart + string("---") +  
                  theActionPart + string(3, '-')  
                  + dogPartCString;
```

- Strings can be used with **cin** / **cout**

```
string name;  
cin >> name;  
cout << name;
```

- **size()** and **length()** return the **number of chars**

```
string greeting = "hello";  
for (int i = 0; i < greeting.size(); i++)  
{  
    cout << greeting[i] << endl;  
}
```

- The `[]` operator is supported – similar to `[]` for a char array

```
string text = "hello";  
text[1] = 'a';  
cout << text << endl; // hallo
```

- The `+` operator concatenates two strings

```
string helloName = hello + string(" ") + name;  
cout << helloName << endl; // "hello George"
```

- `c_str()` - returns the actual C-string of the **std::string object**



- Two strings can be compared with any comparison operator
  - operators `<`, `<=`, `==`, `>=`, `>` compare the strings lexicographically

```
string s1 = "cat", s2 = "canary";  
if (s1 < s2) cout << s1 << " is before " << s2 << endl;  
else cout << s1 << " is after " << s2 << endl;
```

- **`str.find(search)`**
  - Returns the index of **`search`** in the **`str`**
  - If **`search`** is not found, returns the **`string::npos`** value (**`-1`**)

```
cout << "nar" << " at index " << s1.find("nar") << " in " << s2;
```

- The `find(search, index)` overload takes a start index
  - The search starts from that index

```
string s = "aha";  
cout << s.find("a", 1); // prints 2
```

- We can use this to search all occurrences of a substring

```
string str = "canary";  
int foundIndex = str.find("a");  
while (foundIndex != string::npos)  
{  
    cout << foundIndex << endl;  
    foundIndex = str.find("a", foundIndex + 1);  
}
```

# std::string: Substring

- **substr(index, length)** returns a new string
  - With **length** characters, starting from **index**

```
string text = "abc";  
cout << text.substr(1, 2); // prints bc
```

```
string fullName = "Ivan Ivanov";  
string firstName = fullName.substr(0, 4);  
string lastName = fullName.substr(5, 6);  
cout << firstName << endl; // prints Ivan  
cout << lastName << endl; // prints Ivanov
```



# std::string: Erase and Replace

- **erase(index, length)**

- Changes a string by removing chars
- Removes **length** characters, starting from **index**

```
string text = "abc";  
text.erase(1,2);  
cout << text; // prints a
```

- **replace(index, length, str)**

- Changes a string by replacing
- Characters in **[index, index + length)** replaced by **str**

```
string text = "abc";  
text.replace(1, 2, "cme");  
cout << text; // prints acme
```





# Streams

Reading by Line and File Streams

# Streams

- Streams offer an abstraction over incoming or outgoing data of indefinite length
  - **cin** and **cout** are abstractions of the console input or output
- Streams are **ways of reading / writing data**
- A stream can be constructed for any type of data container as:
  - arrays, strings, memory
  - files, network connections, the keyboard buffer



# std::stringstream

- A stream that works on a string

**#include<sstream>**

- Can **read** data from a string
- Can **write** data to a string
  - There are limited **istringstream/ostringstream** versions that only read or write respectively
- Useful for working on a string "word-by-word"



- `istream` is a limited `stringstream` than only reads
  - If you only want to read, use it instead of `stringstream`
- Initialize `istream` by giving it a `string` to read from

```
string str = "3 -2";  
istream numbersStream(str);
```

- From then on, use the stream just like `cin`

```
int num1, num2;  
numbersStream >> num1 >> num2;  
int sum = num1 + num2;
```



# Writing with `std::ostringstream`

- `ostringstream` is a limited `stringstream` than only writes
- Initialize `ostringstream` like a normal variable

```
ostringstream stream;
```

- Use the stream just like `cout`

```
stream << "The sum is " << num1 + num2 << endl;
```

- To get the string when you're done, call `str()`

```
cout << stream.str();
```

- **getline(stream, targetStr)**
  - Reads an entire line of text until a delimiter **char** (additional parameter) is reached
  - From the provided **stream** and puts it into **targetStr**
  - Avoid mixing **cin>>** and **getline(cin,...)**

```
istringstream in("a word");  
  
string line;  
getline(in, line);  
cout << line << endl; // a word
```

```
istringstream in("a.word");  
  
string line;  
getline(in, line, '.');  
cout << line << endl; // a
```


# Parsing Numbers from a Line

- `getline()` already gives us the line as a string
- Streams allow us to **read strings or numbers separated by spaces**
- How do we know when to stop?
  - Streams can be used as a **bool** value
  - A stream is **true** if it still has something to read
  - A stream is **false** if the input ended or if there was an error



# Parsing Numbers from a Line

- Read the line from **cin** into a **string** with **getline()**
- Create an **istringstream** over that **string**
- Read numbers from the stream while the stream is **true**



```
string line;  
getline(cin, line);  
istringstream lineStream(line);  
int numbers[100];  
int currentNumber;  
int count = 0;  
while (lineStream >> currentNumber)  
{  
    numbers[count++];  
}
```

**#include<fstream>**

- **ifstream** is for reading
- **ofstream** is for writing
- Text reading / writing with same operators, functions, concepts
  - **<<** for writing
  - **>>** for reading
  - **getline()** reads line, etc.
  - Can be used as **bool** just like **cin**, **cout** and **stringstream**

- Declare the stream and open the file

- Input streams expect the **file to exist**

```
ifstream input;  
input.open("input.txt");  
int a, b;  
input >> a >> b;  
input.close();
```

- Output streams **create or overwrite the file** on opening

```
ofstream output;  
output.open("output.txt");  
output << a + b << endl; output.close();
```

- Declaration and opening can be shortened

```
ifstream input("input.txt");  
int a, b;  
input >> a >> b;  
input.close();
```

```
ofstream output("output.txt");  
  
output << a + b << endl;  
output.close();
```

- `close()` is automatically called when stream goes out of scope
- To make an output stream append instead of overwrite:  
`ofstream output("output.txt", fstream::app);`

- Text is a sequence of bytes interpreted by special rules
- Two standard ways of working with text:
  - `std::string` is the way for working with text
  - C-Strings (`char` arrays) are the legacy C approach
- Streams are abstractions for writing or reading data





# SoftUni Diamond Partners



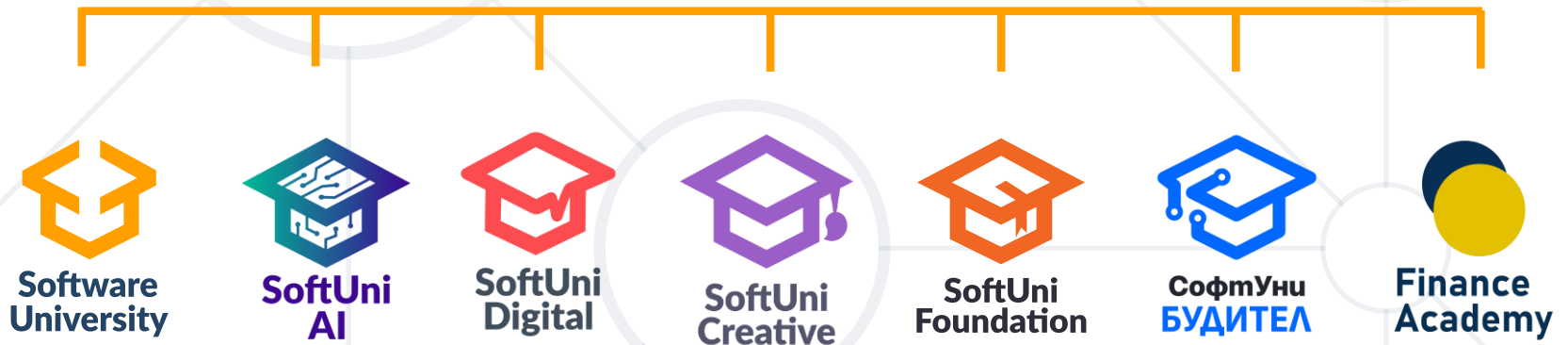
THE CROWN IS YOURS



# Questions?



SoftUni



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>

