

Chapter 2:

String과 String View 다루기

Mijin An

meeeeeejin@gmail.com



C-Style String

myString 'H' | 'E' | 'L' | 'L' | '0' | '\0'

- String의 마지막에 NUL 문자 (\0)를 붙여서 string이 끝났음을 표현
- C++에서는 <cstring> header file을 통해 C의 string 연산에 접근 가능
 - 대체로 메모리 할당 기능을 제공하지 않음
- `strlen()` : string에 담긴 실제 문자 수만 return
 - '\0'을 포함한 길이를 return 하면, 여러 string을 합칠 시 메모리 공간에 딱 맞게 계산하기 어려움
- `sizeof()`: 데이터 타입이나 변수의 크기를 return

sizeof() vs. strlen()

Q. s1, s2, s3, s4 값은 각각 얼마일까?

```
char text1[] = "abcdef";  
size_t s1 = sizeof(text1);  
size_t s2 = strlen(text1);
```

```
const char* text2 = "abcdef";  
size_t s3 = sizeof(text2);  
size_t s4 = strlen(text2);
```

sizeof() vs. strlen()

Q. s1, s2, s3, s4 값은 각각 얼마일까?

```
char text1[] = "abcdef";  
size_t s1 = sizeof(text1); // 7  
size_t s2 = strlen(text1); // 6
```

```
const char* text2 = "abcdef";  
size_t s3 = sizeof(text2); // 8  
size_t s4 = strlen(text2); // 6
```

String Literal

```
cout << "hello" << endl;
```

- 변수에 담지 않고 곧바로 값으로 표현한 string
- 내부적으로 메모리의 **읽기 전용 영역**에 저장
 - 같은 string literal이 코드에 여러 번 나오면 컴파일러는 그중 한 string에 대한 reference를 재사용하여 메모리 절약 → **Literal Pooling**
- String literal을 변수에 **대입** 가능; But, 여러 곳에서 공유할 수 있기 때문에 **위험**
 - String literal을 수정하는 동작에 대해서는 명확한 정의 없음 → 안전하게 const 사용!

```
char* ptr = "hello"; // 변수에 string literal 대입  
ptr[1] = 'a';        // 결과 예측 불가
```

```
const char* ptr = "hello"; // 변수에 string literal 대입  
ptr[1] = 'a';              // 읽기 전용 메모리에 값을 써 에러 발생
```

Raw String Literal

```
const char* str = "Line 1\nLine 2";  
const char* str = R"(Line 1  
Line 2)";
```

- 여러 줄에 걸쳐 작성한 string literal: R" (...)"
- 인용 부호나 escape sequence (\t, \n)를 일반 텍스트로 취급
-)" 문자를 추가하려면 **extended raw string literal** 구문으로 표현:
R"delimiter-char-seq(char-seq)delimiter-char-seq"

```
const char* str = R"(Embedded )" characters);  
const char* str = R"-(Embedded )" characters)-";
```

- DB query, 정규표현식, 파일 경로 등을 쉽게 표현 가능

C++ std::string Class (1)

```
string A("12");  
string B("34");  
string C;  
C = A + B; // "1234"
```

- basic_string 클래스 템플릿의 인스턴스
 - std namespace에 속하며 <string> header에 정의
- **메모리 할당** 작업을 처리해주는 기능이 더 들어있음
- string은 실제로는 클래스지만 마치 기본 타입인 것처럼 사용

C++ std::string Class (2)

```
string myString = "hello";
myString += ", there";
string myOtherString = myString;
if (myString == myOtherString) {
    myOtherString[0] = 'H';
}
cout << myString << endl;           // hello, there
cout << myOtherString << endl;      // Hello, there
```

- String을 할당하거나 크기를 조절하는 코드가 흩어져 있어도 **메모리 leak 발생하지 않음**
 - string 객체는 **stack 변수**로 생성
- 연산자를 원하는 방식으로 사용
 - ==, !=, <와 같은 연산자를 오버로딩해서 string에 적용 가능

C++ std::string Literal

```
auto string1 = "Hello World";    // const char* 타입  
auto string2 = "Hello World"s;   // std::string 타입
```

- const char* 로 처리
- 표준 사용자 정의 literal s를 사용하면 string literal을 std::string으로 변환 가능

High-Level Numeric Conversion: Number → String

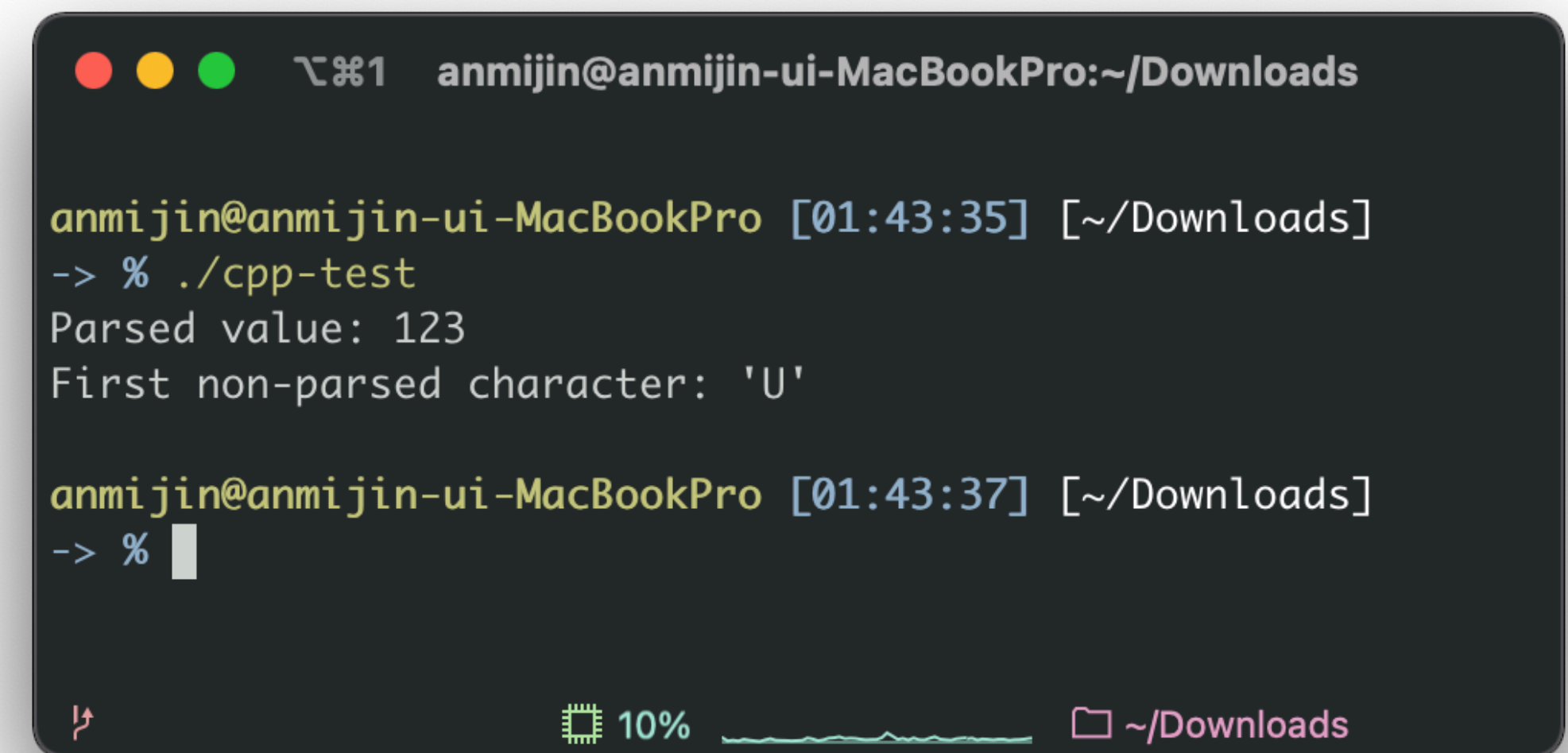
- std namespace는 숫자와 string을 쉽게 변환할 수 있도록 다양한 **helper** 함수 제공
 - 메모리 할당 작업도 처리
 - Number → String
 - `string to_string(int val);`
 - `string to_string(unsigned val);`
 - `string to_string(long val);`
 - `string to_string(unsigned long val);`
 - `string to_string(long long val);`
 - `string to_string(unsigned long long val);`
 - `string to_string(float val);`
 - `string to_string(double val);`
 - `string to_string(long double val);`

High-Level Numeric Conversion: String → Number

- std namespace는 숫자와 string을 쉽게 변환할 수 있도록 다양한 **helper** 함수 제공
 - 메모리 할당 작업도 처리
 - String → Number
 - str: 변환하려는 원본 string
 - idx: 아직 변환되지 않은 부분의 맨 앞에 있는 문자의 인덱스
 - base: 변환할 수의 밑 (기수, 기저)
- `int stoi(const string& str, size_t *idx=0, int base=10);`
- `long stol(const string& str, size_t *idx=0, int base=10);`
- `unsigned long stoul(const string& str, size_t *idx=0, int base=10);`
- `long long stoll(const string& str, size_t *idx=0, int base=10);`
- `unsigned long long stoull(const string& str, size_t *idx=0, int base=10);`
- `float stof(const string& str, size_t *idx=0);`
- `double stod(const string& str, size_t *idx=0);`
- `long double stold(const string& str, size_t *idx=0);`

High-Level Numeric Conversion: String → Number

```
1  #include <iostream>
2  #include <string>
3
4  int main() {
5      using std::cout;
6      using std::endl;
7      using std::string;
8
9      const string toParse = " 123USD";
10     size_t index = 0;
11
12     int value = stoi(toParse, &index);
13     cout << "Parsed value: " << value << endl;
14     cout << "First non-parsed character: '" << toParse[index] << "'" << endl;
15 }
```



A terminal window with a dark background and light-colored text. The window title is "anmijin@anmijin-ui-MacBookPro:~/Downloads". The prompt is "anmijin@anmijin-ui-MacBookPro [01:43:35] [~/Downloads]". The user enters the command "-> % ./cpp-test". The output is "Parsed value: 123" and "First non-parsed character: 'U'". The user enters another command "-> %" and a space character. The terminal shows a cursor at the end of the second command. The bottom of the terminal shows a battery icon at 10% and a folder icon for ~/Downloads.

```
anmijin@anmijin-ui-MacBookPro [01:43:35] [~/Downloads]
-> % ./cpp-test
Parsed value: 123
First non-parsed character: 'U'

anmijin@anmijin-ui-MacBookPro [01:43:37] [~/Downloads]
-> % 
```

Low-Level Numeric Conversion

- C++17부터 `<charconv>` header를 통해 low-level 변환 제공
- **메모리 할당은 하지 않기 때문에 호출 시 버퍼를 할당하여 사용**
- **고성능 및 locale-independent를 위해 튜닝돼 처리 속도 매우 빠름**
 - 숫자 데이터와 사람이 읽기 좋은 포맷 (JSON, XML 등) 간 변환 작업을 빠르게 처리할 시 사용

Low-Level Numeric Conversion: Number → String

- Integer → String

```
to_chars_result to_chars(char* first, char* last, IntegerT value, int base = 10);
```

```
struct to_chars_result {  
    char* ptr;  
    errc ec;  
};
```

- Float/Double → String

```
to_chars_result to_chars(char* first, char* last, FloatT value,  
| | | | | chars_format format, int precision);
```

```
enum class chars_format {  
    scientific,           // (-)d.ddde ±dd  
    fixed,               // (-)ddd.ddd  
    hex,                 // (-)h.hhhp ±d (0x는 적지 않는다)  
    general = fixed | scientific  
};
```

- 💡 general을 적용하면 둘 중 소수점 왼쪽 숫자의 길이가 짧은 값으로 변환
- 💡 format에 precision을 지정하지 않으면 주어진 format에서 가장 짧은 형태 사용
- 💡 precision의 최댓값은 6자리

Low-Level Numeric Conversion: String → Number

```
from_chars_result from_chars(const char* first, const char* last,  
                             IntegerT& value, int base = 10);  
  
from_chars_result from_chars(const char* first, const char* last,  
                             FloatT& value,  
                             chars_format format = chars_format::general);  
  
struct from_chars_result {  
    const char* ptr;  
    errc ec;  
};
```

C++ std::string_view Class

- C++17 이전에는 읽기 전용 string을 받는 함수의 매개변수 타입 결정이 어려웠음
 - `const char*` → `std::string`의 경우 `c_str()`나 `data()`를 이용해 변환 필요
 - 이 경우, `string`의 객체지향 속성 및 helper 함수 활용 불가능
 - `const std::string&` 사용?
 - String literal 전달 시 컴파일러는 그 복사본이 담긴 `string` 객체를 생성해서 함수로 전달
→ 오버헤드 존재
- C++17부터 `std::string_view` 클래스 사용!
- `basic_string_view` 클래스 템플릿의 인스턴스
 - `std` namespace에 속하며 `<string_view>` header에 정의
 - `const string&` 대신 사용 가능
 - string을 복사하지 않아 오버헤드 없음

C++ std::string_view Class: Example

- Pass-by-value

```
string_view extractExtension(string_view fileName) {  
    return fileName.substr(fileName.rfind('.'));  
}  
  
void printResults() {  
    string fileName = R"(/home/mijin/myfile.ext)";  
    cout << "C++ string: " << extractExtension(fileName) << endl;  
}
```

- string_view는 string 생성은 하지 않음
 - string 생성자를 직접 호출하거나 string_view::data()로 생성

```
void handleExtension(const string& extension) { /* ... */ }  
  
handleExtension(extractExtension("my-file.txt").data());  
handleExtension(string(extractExtension("my-file.txt")));
```

std::string_view Literal

- 표준 사용자 정의 literal인 sv를 사용해 string literal을 std::string_view로 변환

```
auto sv = "My string_view"sv;
```

Reference

[1] Marc Gregoire, “Professional C++, 5th Edition”, Wiley, 2021