

C++プログラミングII

第12回 string と正規表現

岡本秀輔

成蹊大学理工学部

string と探索アルゴリズム

std::string と探索アルゴリズム

- ▶ string は添字と std::string::npos を使用する
- ▶ コンテナではないが類似メンバ関数を持つ
 - ▶ begin, end, size, clear, swap, push_back, pop_back, insert, など
- ▶ STL アルゴリズムが適用可能
 - ▶ begin, end で得たイテレータに互換性がある
 - ▶ ランダム・イテレータ
- ▶ 独自のメンバ関数で高速な処理を実現

探索対象	std::string	STL アルゴリズム
最初に現れる要素	find	find
最後に現れる要素	rfind	find と逆イテレータ
最初に現れる部分範囲	find	search
最後に現れる部分範囲	rfind	find_end
部分範囲の高々 1 つが同じ最初	find_first_of	find_first_of
部分範囲の高々 1 つが同じ最後	find_last_of	find_first_of と逆イテレータ
最初に現れる n 個連続部分		search_n

部分文字列

▶ イテレータとメンバ関数で部分文字列の作成

```
#include <iostream>
using std::cout, std::string;
int main() {
    string s{"hello, world"};
    auto b{s.begin()}, e{s.end()};

    string t(b, b+6); // 波括弧も可
    cout << t          << " "
         << string(b+7, e) << " " // Ctor
         << s.substr(7)   << " " // 先頭添字
         << s.substr(7,5)   // 先頭添字, 文字数
         << "\n"; // hello, world world world
}
```

std::find とメンバ関数 find

- ▶ 1文字を探す
- ▶ メンバ関数 find は添字を返す

```
#include <algorithm>
#include <iostream>
using std::cout, std::string;
int main() {
    string t{"I am running now."};
    // STL アルゴリズム
    auto it {std::find(t.begin(), t.end(), 'n')};
    if (it != t.end())
        cout << "found at " << it-t.begin() << "\n";

    // メンバ関数
    size_t idx { t.find('n') };
    if (idx != string::npos)
        cout << "found at " << idx << "\n";
}
```

std::find とメンバ関数 rfind

- ▶ 後ろから 1 文字を探す (reverse の r)
- ▶ 逆イテレータよりもメンバ関数を使用した方が良い

```
#include <algorithm>
#include <iostream>
using std::cout, std::string;
int main() {
    string t{"I am running now."};
    // STL アルゴリズム
    auto it {std::find(t.rbegin(), t.rend(), 'n')};
    if (it != t.rend())
        cout <<"found at "
        << (t.size()-1)-(it-t.rbegin()) <<"\n";

    // メンバ関数
    size_t idx { t.rfind('n') };
    if (idx != string::npos)
        cout <<"found at " << idx <<"\n";
}
```

std::search とメンバ関数 find

- ▶ std::search は汎用なので指定が冗長に見える

```
#include <algorithm>
#include <iostream>
using std::cout, std::string;
int main() {
    string t{"I am running now."}, w{"ing"};
    // STL アルゴリズム
    auto it {std::search(t.begin(), t.end(),
                        w.begin(), w.end()) };
    if (it != t.end())
        cout <<"found at " << it-t.begin() <<"\n";

    // メンバ関数
    size_t p { t.find(w) };
    if (p != string::npos)
        cout <<"found at " << p <<"\n";
}
```

std::find_end とメンバ関数 rfind

- ▶ t の後ろ側から部分範囲 w を探す

```
#include <algorithm>
#include <iostream>
using std::cout, std::string;
int main() {
    string t{"NumPy is a Python library"}, w{"Py"};
    // STL アルゴリズム
    auto it {std::find_end(t.begin(), t.end(),
                           w.begin(), w.end())};
    if (it != t.end())
        cout <<"found at " << it-t.begin() <<"\n";

    // メンバ関数
    size_t p { t.rfind(w) };
    if (p != string::npos)
        cout <<"found at " << p <<"\n";
}
```


std::find_first_of とメンバ関数 find_first_of

- ▶ t から s のどれかの文字を探す

```
#include <algorithm>
#include <iostream>
using std::cout, std::string;
int main(){
    string t{"hello, world"}, s{"orz"};
    // STL アルゴリズム
    auto it {std::find_first_of(t.begin(), t.end(),
                                s.begin(), s.end())};
    if (it != t.end())
        cout << it-t.begin() << ":" << *it << "\n";

    // メンバ関数
    size_t pos = t.find_first_of(s);
    if (pos != string::npos)
        cout << pos << ":" << t[pos] << "\n";
}
```

STL アルゴリズムによる単語抽出

- ▶ 区切り文字を指定して単語を抽出
- ▶ STL アルゴリズムのみで実装できるが煩雑

```
#include <algorithm>
#include <iostream>
#include <iterator>
using std::cout, std::string;
int main() {
    string t{" Hooray, it's nice!! I like it."},
           delim{".,?! "}; // delimiters
    auto head{t.begin()}, e{t.end()},
         b2{delim.begin()}, e2{delim.end()};
    while (true) {
        auto tail {std::find_first_of(head, e, b2, e2)};
        if (tail == e) break;
        if (head != tail)
            cout << string(head, tail) << "\n";
        head = std::next(tail); // 区切り文字の次から
    }
}
```

string メンバ関数による単語抽出

- ▶ pos2 string::find_first_of(str, pos1)
 - ▶ str: 探したい文字群, pos1: 調べる先頭位置
 - ▶ pos2: 見つけた文字の位置
- ▶ 前のスライドと同じ内容

```
#include <iostream>
using std::cout, std::string;
int main() {
    string t{" Hooray, it's nice!! I like it."},
           delim{".,?! "};
    size_t head {0};
    while (true) {
        size_t tail {t.find_first_of(delim, head)};
        if (tail == string::npos) break;
        if (head != tail)
            cout << t.substr(head, tail-head) << "\n";
        head = tail+1; // 区切り文字の次から
    }
}
```

正規表現

正規表現とは

- ▶ 文字列のグループ（集合）を指定する方法
 - ▶ 複数の文字列をグループ化する
 - ▶ 文字列を探す際の条件指定
- ▶ 文字の指定の例
 - ▶ a や b などのそのままの 1 文字
 - ▶ アルファベットに属する 1 文字
 - ▶ 数字 1 文字
 - ▶ ある範囲の 1 文字 (0~7 や a~z など)
 - ▶ 改行文字以外の任意の 1 文字 (どれでも良い)
- ▶ やり方：特定の文字に特殊な意味を持たせる
 - ▶ `^ $ \ . * + ? () [] { } |` は特殊な意味を持つ
 - ▶ `\n` や `\w` など、バックスラッシュ+文字の形で、特殊な意味を表したり、逆に特殊な意味を無効化したりする。

std::ECMAScript 構文

- ▶ 正規表現にはいろいろな種類がある
- ▶ C++11 の正規表現は JavaScript 言語の規格に沿う
- ▶ その他の種類の正規表現もオプションにより指定可能
- ▶ 単純な例

正規表現	マッチする文字列	備考
ab	ab	文字列そのものの指定
a b	a、b	二者の選択
a*	a、aa、aaa	a の 0 回以上の繰り返し
a(a b)*	a、aa、ab、aaa、 aab、aababa、...	a で後に a か b が 0 回以上続く
(a b)*c	c、ac、bc、 aac、abc、...	a と b が 0 回以上続き c が現れる

正規表現ルール1

▶ 1 文字の指定パターン

指定	マッチする文字
----	---------

指定	マッチする文字
\t	水平タブ
\n	改行文字 (LF:Line Feed, 0xA)
\r	キャリッジリターン文字 (CR:Carriage Return, 0xD)
\0	ヌル文字
.	改行文字以外の 1 文字 (\r や \n 以外)
\d	10 進数の数字 (digit)
\D	10 進数の数字以外の文字 (大文字は補集合)
\s	ホワイトスペース (改行を含む)
\S	ホワイトスペース以外の文字
\w	英数字または下線文字
\W	英数字でも下線文字でもない文字

正規表現ルール2

▶ 範囲指定の1文字パターン

- ▶ 角括弧を使ってマッチする1文字のクラスを指定
- ▶ `[^範囲]` は補集合の意味

指定	マッチする文字
<code>[abc]</code>	abc のどれかの1文字
<code>[a-z]</code>	a から z のどれかの1文字
<code>[\t\n\r]</code>	スペース文字, <code>\t</code> , <code>\n</code> , <code>\r</code> のどれか1文字
<code>[^abc]</code>	abc 以外の1文字

正規表現ルール3

▶ 直前パターンの繰り返しの指定

指定	マッチする文字/パターン
*	0 またはそれ以上の繰り返し
+	1 回以上の繰り返し
?	あってもなくても良い (0 or 1)
{n}	ちょうど n 回の繰り返し
{n,}	n 回以上の繰り返し
{n,m}	n 回以上 m 回以下の繰り返し

▶ 貪欲 ルール：デフォルトで最長の繰り返しパターン

▶ 非貪欲ルール：?をつけると最短の繰り返し回数

- ▶ 例：入力 aaa に対して、
a+は aaa にマッチし
a+?は a にマッチする

正規表現ルール4

▶ グループ化

- ▶ グループを * などの繰り返しの対象に使用可能
- ▶ 必要ならばマッチしたサブパターンを後で利用可能

(サブパターン)	グループ化して後で利用
(?: サブパターン)	グループ化するのみ

▶ 表明 (アサーション)

- ▶ パターンの前後の条件を指定する

^	続くパターンは先頭 (補集合ではない)
\$	この前のパターンは末尾
\b	単語の区切り

▶ 二者択一

	両側の二つのパターンどちらか
--	----------------

正規表現の例

正規表現	マッチする文字
<code>a.*b</code>	ab, anb, accab など (a から b の並び)
<code>a{3}</code>	aaa, ちょうど 3 文字の a
<code>a{3,5}</code>	aaa, aaaa, aaaaa のどれか
<code>[a-zA-Z_]\w*</code>	c++ の変数/関数などの名前
<code><[a-z]+>.*</[a-z]+></code>	XML(html) の開きタグと閉じタグ
<code><([a-z]+)>.*</\1></code>	対応する開きタグと閉じタグ
<code>[^xyz]</code>	xyz 以外の文字
<code>^xyz</code>	先頭にある xyz
<code>xyz\$</code>	末尾にある xyz
<code>\bxyz</code>	文字列の途中ではない xyz
<code>abc xyz</code>	abc または xyz のどちらか
<code>\b\w+\.(?:\w){3}\b</code>	3 文字拡張子の英数字ファイル名

Raw 文字列

Raw 文字列

- ▶ C++の文字列リテラルでは\文字が特殊な意味を持つ
- ▶ 正規表現の\wを指定するには"\\w"と書く必要がある
 - ▶ 複雑な正規表現は\が多数並ぶ

Raw 文字列とは (Raw: 生の, 加工していない):

- ▶ 指定した文字をそのまま扱う別の文字列リテラル
- ▶ 改行もそのまま文字列の一部となる
- ▶ R"(文字列)"とする。
 - ▶ つまり R"(と)"で文字列をくくる。
- ▶ 文字列にラベルを付けても良い
 - ▶ R"ラベル (文字列) ラベル"の形式
 - ▶ エディタで検索しやすくなり、)"も指定できる

```
"[ \\t\\n]*"           // 通常の文字列指定
R"([ \\t\\n]*)"         // Raw 文字列
R"var([a-zA-Z_]\\w*)var" // var がラベル
```

C++11 の正規表現

省略

- ▶ この後のコードでは以下を仮定する。

```
#include <iostream>
#include <regex>
using std::cout, std::string;
```

概要

- ▶ `<regex>` ヘッダファイル
- ▶ 正規表現指定クラス: `std::regex`
 - ▶ 例: `std::regex r{"a[bc]c"};`
- ▶ 文字列 `s` からパターンを探す関数
 - ▶ 対象全体に対するマッチ:
 - ▶ `std::regex_match(s, r)` : `r` がマッチしたか?
 - ▶ `std::regex_match(s, m, r)`: マッチの詳細を `m` に
 - ▶ マッチの探索 (部分一致):
 - ▶ `std::regex_search(s, r)` : `r` のマッチを探す
 - ▶ `std::regex_search(s, m, r)`: マッチの詳細を `m` に

```
int main() {  
    string src{"abc"};    // 検索対象  
    std::regex reg{"a[bc]c"}; // 正規表現  
    cout << std::boolalpha  
          << std::regex_match(src, reg) << "\n"; // true  
}
```


regex_match() と regex_search()

- ▶ マッチの有無を bool 型で返す
- ▶ regex_match() は全体への一致を求める
- ▶ regex_search() は部分一致も許容する

```
int main() {  
    std::regex r{"a.*c"}; // 正規表現: 先頭 a 末尾 c  
    cout<< std::boolalpha;  
    cout<< std::regex_match("ac", r)<<"\n";  
    cout<< std::regex_match(" aaac", r)<<"\n"; //false  
    cout<< std::regex_match("ab ac", r)<<"\n";  
    cout<< std::regex_match("abc ", r)<<"\n"; //false  
    cout<<"---\n";  
    cout<< std::regex_search("ac", r)<<"\n";  
    cout<< std::regex_search(" aaac", r)<<"\n";  
    cout<< std::regex_search("ab ac", r)<<"\n";  
    cout<< std::regex_search("abc ", r)<<"\n";  
}
```

改行を含む例

- ▶ `.` は改行文字を含まないパターン
- ▶ `regex_match()` でも正規表現に周辺のパターンを含めれば同じ (改行文字は特別扱いが必要)

```
int main() {  
    string s{" ab\ncdefg "};  
    cout << std::boolalpha;  
  
    std::regex r1{"cde"};  
    cout << std::regex_search(s, r1) << "\n"; //true  
  
    std::regex r2{R"(. *cde.*)"};  
    cout << std::regex_match(s, r2) << "\n"; //false  
  
    std::regex r3{R"((.|\n)*cde.*)"};  
    cout << std::regex_match(s, r3) << "\n"; //true  
}
```

結果オブジェクト

- ▶ `std::smatch` オブジェクトでマッチの詳細が得られる

```
void test(string s, string reg) {
    cout << s << ", " << reg << ":\n";
    std::regex r{reg};
    std::smatch m;
    if (std::regex_match (s, m, r))
        cout << "    match : m[0]=" << m[0] << "\n";
    if (std::regex_search(s, m, r))
        cout << "    search: m[0]=" << m[0] << "\n";
    // string を得るには m[0].str() を指定する
}

int main() {
    test("get_X", "get|get_X");
    test("get_XY", "get|get_X");
    test("get_X", "get.*");
}
```

出力結果

- ▶ 同一の正規表現でも match と search で結果が異なる
- ▶ search は .* に対して貪欲ルールで最大一致をとる

```
get_X, get|get_X:
    match : m[0]=get_X
    search: m[0]=get
get_XY, get|get_X:
    search: m[0]=get
get_X, get.*:
    match : m[0]=get_X
    search: m[0]=get_X
```

std::smatch の詳細

std::smatch m; に対して

メンバ関数	意味
m.empty()	マッチの有無
m.prefix()	マッチする前の文字列
m.suffix()	マッチ後の文字列
m.size()	m[n] のサイズ (マッチしなければ 0)
m[0]	マッチした文字列
m[n]	グループ化した部分 ($n > 0$)
m.position(n)	入力文字列での m[n] の位置

```
int main() {
    string data { "abc<h1>title</h1>xyz" };
    cout <<"data: " << data <<"\n";
    std::regex r{ R"(<(\w+)>(.*)</(\1)>)" };
    std::smatch m;
    std::regex_search(data, m, r);
    if (!m.empty()) {
        cout <<"m.prefix():"<< m.prefix() <<"\n"
              <<"m.suffix():"<< m.suffix() <<"\n";
        for (size_t i = 0; i < m.size(); i++) {
            cout <<"m["<< i <<"]:"<< m[i] <<" , "
                  <<"m.position("<< i <<"):"
                  << m.position(i) <<"\n";
        }
    }
}
```

正規表現の確認

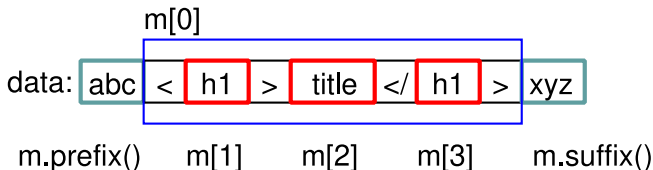
正規表現：`<(\w+)>(.*?)</(\1)>`

- ▶ `<(\w+)>`
 - ▶ XML の開きタグに対応
 - ▶ `<`で始まり, 1 文字以上の英数文字, `>`が続く
 - ▶ 1 文字以上の英数字がマッチしたら記憶 (番号 1)
- ▶ `(.*?)`
 - ▶ 0 文字以上の改行以外の文字
 - ▶ これもマッチしたら記憶 (番号 2)
- ▶ `</(\1)>`
 - ▶ XML の閉じタグに対応
 - ▶ `</`の文字列, 番号 1 で記憶した文字列, `>`の文字
 - ▶ これもマッチしたら記憶 (番号 3)

出力結果

- ▶ position は文字列 data の添字に対応する

```
data: abc<h1>title</h1>xyz  
m.prefix():abc  
m.suffix():xyz  
m[0]:<h1>title</h1>, m.position(0):3  
m[1]:h1, m.position(1):4  
m[2]:title, m.position(2):7  
m[3]:h1, m.position(3):14
```



貪欲と非貪欲マッチ

- ▶ 直前パターンの繰り返しは貪欲と非貪欲の2種
- ▶ 貪欲マッチは最長の文字列を探す (デフォルト)
- ▶ 非貪欲マッチは最短の結果となる ('?' の指定)

```
void test(string s, std::regex r) {
    std::smatch m;
    if (std::regex_search(s, m, r) && m.size() > 1)
        cout << m[1] << "\n";
}

int main() {
    std::regex g0{"(a.*)"}, n0{"(a.*?)"};
    test("aaabbbb", g0); // aaabbbb
    test("aaabbbb", n0); // a
    std::regex g1{"(a.*c)"}, n1{"(a.*?c)"};
    test("aacacacdgc", g1); // aacacac
    test("aacacacdgc", n1); // aac
}
```

string イテレータの利用

- ▶ `cbegin()`, `cend()` は `const` イテレータ
- ▶ `m.suffix().first` はマッチの次の文字のイテレータ

```
int main() { // d は改行を含む文字列
    string d{ R"(
<student><first>Momoko</first>
<last>Seikei</last></student>)" };
    cout << d << "\n";
    std::regex r{R"abc(<(\w+)>(.*)</\1>)abc"};
    std::smatch m;
    for (auto pos{d.cbegin()};
        std::regex_search(pos, d.cend(), m, r);
        pos = m.suffix().first) {
        cout << "-----\n"
            << "match: " << m[0] << "\n"
            << "tag:   " << m[1] << "\n"
            << "value: " << m[2] << "\n";
    }
}
```

出力結果

- ▶ `regex_search()` は改行を除き最大一致をとる

```
<student><first>Momoko</first>  
<last>Seikei</last></student>
```

```
-----  
match: <first>Momoko</first>  
tag:   first  
value: Momoko
```

```
-----  
match: <last>Seikei</last>  
tag:   last  
value: Seikei
```

<student>はマッチしない

データに改行がない場合

- ▶ 改行のない対象文字列では全体がマッチする

```
<A><B>Momoko</B><C>Seikei</C></A>
```

```
-----
```

```
match: <A><B>Momoko</B><C>Seikei</C></A>
```

```
tag:   A
```

```
value: <B>Momoko</B><C>Seikei</C>
```

- ▶ 正規表現の変更で対応可能:<(\w+)>([<]*)</\1>

```
<A><B>Momoko</B><C>Seikei</C></A>
```

```
-----
```

```
match: <B>Momoko</B>
```

```
tag:   B
```

```
value: Momoko
```

```
-----
```

```
match: <C>Seikei</C>
```

```
tag:   C
```

```
value: Seikei
```

データに改行がない場合: 失敗例

- ▶ `<(\w+)>(.*?)</\1>`では非貪欲でも対応できない

```
<A><B>Momoko</B><C>Seikei</C></A>
```

```
-----
```

```
match: <A><B>Momoko</B><C>Seikei</C></A>
```

```
tag:    A
```

```
value: <B>Momoko</B><C>Seikei</C>
```

- ▶ `<A>`見つけた後に非貪欲でも``が見つかる
- ▶ 最も左の文字からマッチを探すため (最左マッチ)
- ▶ `Momoko`にマッチするには`<A>`のマッチなしが必要

smatch用イテレータの利用

- ▶ 内部で `std::regex_search()` を呼ぶ
- ▶ イテレータは `std::smatch` を指している
- ▶ プログラムが少し短くなる

```
int main() {
    string d {R"(<A><B>Momoko</B><C>Seikei</C></A>)"};
    cout << d << "\n";
    std::regex r{R"(<(\w+)>([^\>]*)</\1>)"};
    std::sregex_iterator pos{d.cbegin(), d.cend(), r};
    std::sregex_iterator end;
    for ( ; pos != end; ++pos){
        auto m{*pos}; // std::smatch
        cout << "-----\n"
             << "match: " << m[0] << "\n"
             << "tag:    " << m[1] << "\n"
             << "value: " << m[2] << "\n";
    }
}
```

置き換え:std::regex_replace

- ▶ \$1, \$2 はグループ化文字列

```
int main() {  
    string d {R"(<A><B>Momoko</B><C>Seikei</C></A>)"};  
    cout << d << "\n";  
    std::regex before{R"(<(\w+)>([^\>]*)</\1>)"};  
    string      after {"<$1 val=\" $2\" />"};  
    string rst1, rst2;
```

// 関数の戻り値

```
rst1 = std::regex_replace(d, before, after);  
cout << rst1 << "\n";
```

// 範囲を明示して 1 文字ずつ rst2 へ

```
std::regex_replace(std::back_inserter(rst2),  
                  d.cbegin(), d.cend(), before, after);  
cout << rst2 << "\n";  
}
```

出力結果

- ▶ 結果はどちらも同じ
- ▶ XML/HTML の記法
 - ▶ `<tag />`は閉じタグを省略する書き方
 - ▶ `<tag attrib="1"/>`は `attrib` という属性値の指定
 - ▶ 属性が複数ある場合にはスペース文字で区切って並べる

```
<A><B>Momoko</B><C>Seikei</C></A>
```

```
<A><B val="Momoko" /><C val="Seikei" /></A>
```

```
<A><B val="Momoko" /><C val="Seikei" /></A>
```


トークンの取得:std::sregex_token_iterator

- ▶ 文字列（トークン）を取り出すイテレータ
- ▶ コンストラクタの引数
 - ▶ 第 1、2 で対象範囲, 第 3 に正規表現, 第 4 引数は下記
 - 0 マッチした文字列
 - 1 マッチしなかった文字列
 - n 正規表現内のグループ n の文字列

```
int main() {
    string t{"abc bcd cdef defgh ijk xyz"};
    std::regex r{R"(\w+)"};

    auto b{t.cbegin()}, e{t.cend()};
    std::sregex_token_iterator p{b, e, r, 0};
    std::sregex_token_iterator end;

    for ( ; p != end; ++p)
        cout <<"\'" << *p <<"\' ";
    cout <<"\n";
    // 'abc' 'bcd' 'cdef' 'defgh' 'ijk' 'xyz'
}
```

マッチしなかった文字列

- ▶ マッチしたパターンの間にある文字列を取り出せる
 - ▶ つまり、マッチしなかった文字列
- ▶ 区切り文字を正規表現で指定すれば残りが取り出せる

```
int main() {
    string t{"Hooray, it's nice! \n I like it."};
    std::regex sep{R"([.,?!\\s]+)"}; // 注)

    auto b{t.cbegin()}, e{t.cend()};
    std::sregex_token_iterator p{b, e, sep, -1}; // *
    std::sregex_token_iterator end;

    for ( ; p != end; ++p)
        cout <<"\\"<< *p <<"\\" ";
    cout <<"\n";
    // 'Hooray' 'it's' 'nice' 'I' 'like' 'it'
}
```

注) 正規表現の `[]` の中では `~\` 以外の文字は特殊な意味を持たない

グループ化文字列と複数指定

- ▶ 第4引数に1以上を指定するとグループ化した文字列
- ▶ 複数の場合には波括弧でリストを指定する
 - ▶ {0,1}: 全体文字列とグループ1

```
int main() {
    string t{"<A><B>Momoko</B><C>Seikei</C></A>"};
    std::regex tag{R"(<(\w+)>[^\>]*</\1>)" };

    auto b{t.cbegin()}, e{t.cend()};
    std::sregex_token_iterator p{b, e, tag, {0,1}};
    std::sregex_token_iterator end;

    for ( ; p != end; ++p)
        cout <<"\'" << *p <<"\' ";
    cout <<"\n";
    // '<B>Momoko</B>' 'B' '<C>Seikei</C>' 'C'
}
```

日本語の扱いについて

正確に書くと長くなるので大雑把に

日本語処理は難しい

- ▶ char 型は通常 8 ビットで英数記号の文字用
- ▶ std::string は char 型データの並びを扱う
- ▶ ひらがなと漢字を表すには 8 ビットでは足りない
- ▶ 文字集合とその符号化はそれぞれ複数ある
 - ▶ 文字集合とは文字に対する数値の表
 - ▶ 符号化とは文字の数値を具体的なビット列にする方法
- ▶ 現状の日本語の符号化の主流も複数
 - ▶ Shift JIS(の変種)、UTF-8, UTF-16, ...
- ▶ C++ ではそれらの符号化を直接サポートしない
 - ▶ 日本語のプログラムはたいてい環境/OS 依存
 - ▶ 変換のライブラリも。。。.
 - ▶ std::string で日本語が扱えても環境/OS 依存

文字「あ」のコード例

Shift JIS: |0x82|0xA0|

UTF-8 : |0xE3|0x81|0x82|

UTF-16 : |0x30|0x42|

日本語文字列と正規表現

- ▶ `std::regex`, `std::smatch` は `std::string` で使える
- ▶ 正規表現で日本語の文字列を表すには...
 - ▶ `wchar_t`: ワイド文字用のデータ型を基本に、
 - ▶ `std::wregex`, `std::wsmatch`, `std::wstring` を使う
 - ▶ 入出力には設定変更（ロケール）が必要
(または文字コードの変換)
 - ▶ ただ、正規表現でどれくらい扱えるかはあやしい
 - ▶ 例 1: `.` (ドット): いわゆる全角文字にもマッチする
 - ▶ 例 2: `\d`: いわゆる全角の 1 2 3 にはマッチしない
- ▶ 正規表現で ASCII の範囲のみ扱うならばなんとかなる
 - ▶ 入出力の文字コードは確認／変換が必要
 - ▶ 日本語の文字列部分を変更しない
 - ▶ 例: 「`<t1>ラベル</t1>`」ならば処理できることが多い