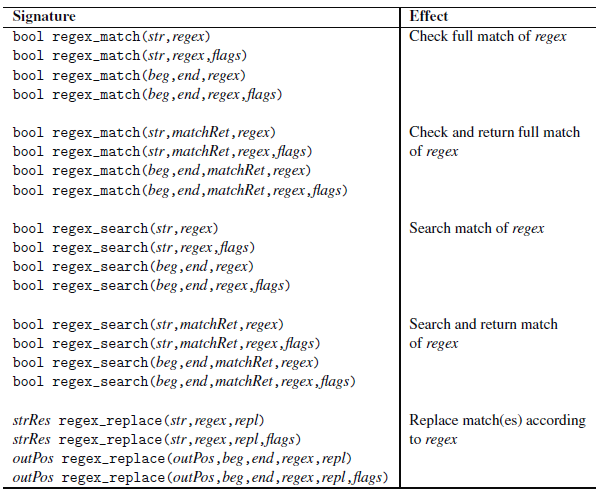
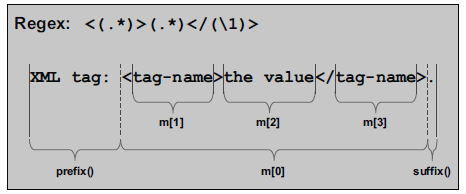
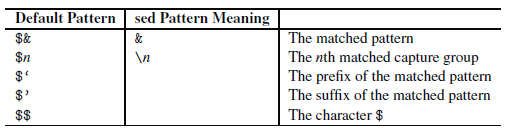
[Regex - C++11, 6 of n](http://blog.csdn.net/cpp_chen/article/details/7984635)

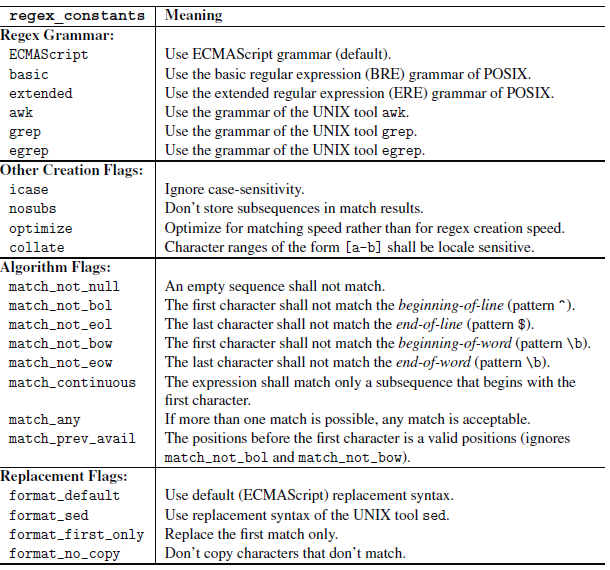
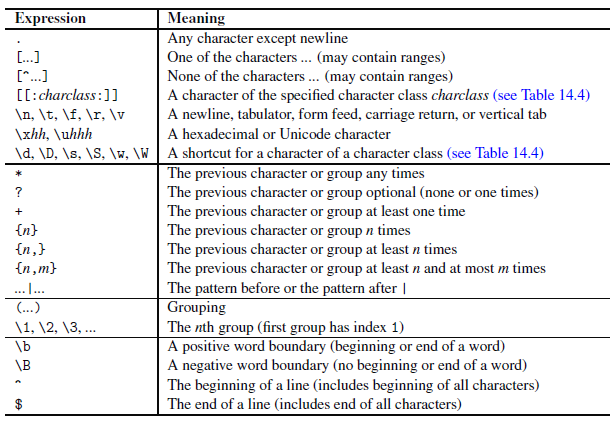
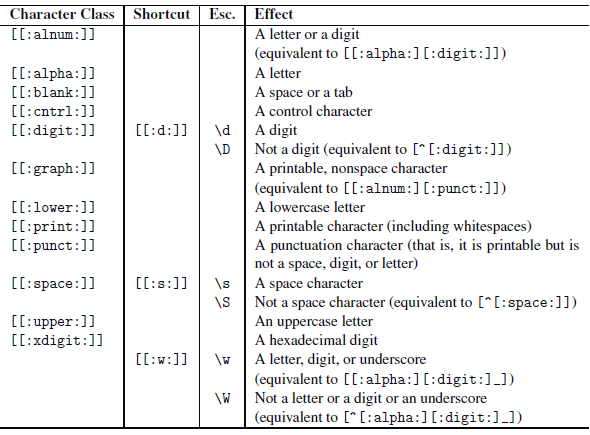
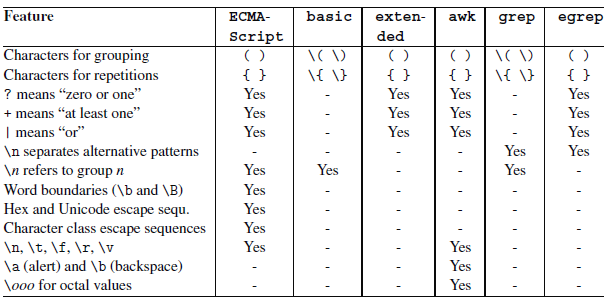
分类： [C++](http://blog.csdn.net/CPP_CHEN/article/category/919478)2012-09-16 18:19 402人阅读 [评论](http://blog.csdn.net/cpp_chen/article/details/7984635#comments)(4) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/cpp_chen/article/details/7984635#report)

**1) Raw string**  
R"delim(...)delim", where delim is a character sequence of at most 16 basic characters except the backslash, whitespace and parentheses.  
**2) Note regex algorithm is greedy  
3) Main interfaces**  
    a. basic\_regex<> template: holds the regular expressionpattern (predefined regex, wregex)  
    b. match\_results<> template: carry back the match results (predefineds/c/ws/wcmatch, s for string, c for const char\*)  
    c. regex\_match(): checks whether the whole character sequence matches a regular expression

    d. regex\_search(): checks whether the character sequence partially matches a regular expression  
     f. regex\_replace(): replace character sequences that match a regular expression.  


**4) Basic matching examples**  
regex reg1("<.\*>.\*</.\*>");  
bool found = regex\_match ("<tag>value</tag>", reg1);  
  
regex reg2("<(.\*)>.\*</\\1>");  // grouping and forward referencing  
found = regex\_match ("<tag>value</tag>", reg2);  
  
// use grep grammer  
regex reg3("<\\(.\*\\)>.\*</\\1>",regex\_constants::grep);  
found = regex\_match ("<tag>value</tag>", reg3);  
  
found = regex\_match ("XML tag: <tag>value</tag>", regex(R"#<(.\*)>.\*</\1>#")); // fails to match  
found = regex\_search ("XML tag: <tag>value</tag>", regex(R"#<(.\*)>.\*</\1>#")); // match

**5) match\_results carry back the detail match result**  
  
match\_results object m has a state which provides the following interface:  
In general, the match\_results object contains:  
– A sub\_match object m[0] for all the matched characters  
– A prefix(), a sub\_match object that represents all characters before the first matchedcharacter  
– A suffix(), a sub\_match object that represents all characters after the last matchedcharacter  
• In addition, for any capture group, you have access to a corresponding sub\_match object m[n].  
• size() yields the number of sub\_match objects (including m[0]).  
• All sub\_match objects are derived from pair<> and have the position of the first character as  
member first and the position after the last character as member second. In addition, str()  
yields the characters as a string, length() yields the number of characters, operator << writes  
the characters to a stream, and an implicit type conversion to a string is defined.  
  
In addition, the match\_results object as a whole provides:  
– member function str() to yield the matched string as a whole (calling str() or str(0))  
or the nth matched substring (calling str(n)), which is empty if no matched substring exists  
(thus, passing an n greater than size() is valid)  
– member function length() to yield the length of the matched string as a whole (calling  
length() or length(0)) or the length of the nth matched substring (calling length(n)),  
which is 0 if no matched substring exists (thus, passing an n greater than size() is valid)  
– member function position() to yield the position of the matched string as a whole (calling  
position() or position(0)) or the position of the nth matched substring (calling  
length(n))  
– member functions begin(), cbegin(), end(), and cend() to iterate over the sub\_match  
objects m[0] to m[n]  
  
**6) regex\_iterator<>, predefinesare s/c/w/wcregex\_iterator**  
Examples:  
string data = "<person>\n"  
                   " <first>Nico</first>\n"  
                   " <last>Josuttis</last>\n"  
                   "</person>\n";  
regex reg("<(.\*)>(.\*)</(\\1)>");  
// iterate over all matches (using a regex\_iterator):  
sregex\_iterator pos(data.cbegin(),data.cend(),reg);  
sregex\_iterator end;  
for ( ; pos != end ; ++pos ) {  
    cout << "match: " << pos->str() << endl;  
    cout << " tag: " << pos->str(1) << endl;  
    cout << " value: " << pos->str(2) << endl;  
}  
// use a regex\_iterator to process each matched substring as element in an algorithm:  
sregex\_iterator beg(data.cbegin(),data.cend(),reg);  
for\_each (beg,end,[](const smatch& m) {  
                               cout << "match: " << m.str() << endl;  
                               cout << " tag: " << m.str(1) << endl;  
                               cout << " value: " << m.str(2) << endl;  
                           });  
  
**7) regex\_token\_iterator<> tokenizer, predefines are s/c/w/wcregex\_token\_iterato**r  
Example:  
string data = "<person>\n"  
                   "<first>Nico</first>\n"  
                   "<last>Josuttis</last>\n"  
                   "</person>\n";  
regex reg("<(.\*)>(.\*)</(\\1)>");  
// iterate over all matches (using a regex\_token\_iterator):  
sregex\_token\_iterator pos(data.cbegin(),data.cend(), // sequence  
                                      reg, // token separator  
                                      {0,2}); // 0: full match, 2: second substring  
sregex\_token\_iterator end;  
for ( ; pos != end ; ++pos ) {  
    cout << "match: " << pos->str() << endl;  
}  
cout << endl;  
string names = "nico, jim, helmut, paul, tim, john paul, rita";  
regex sep("[ \t\n]\*[,;.][ \t\n]\*"); // separated by , ; or . and spaces  
sregex\_token\_iterator p(names.cbegin(),names.cend(), // sequence  
                                   sep, // separator  
                                   -1); // -1: values between separators  
sregex\_token\_iterator e;  
for ( ; p != end ; ++p ) {  
    cout << "name: " << \*p << endl;  
}  
  
**8) regex\_replace**  
string data = "<person>\n"  
                   "<first>Nico</first>\n"  
                   "<last>Josuttis</last>\n"  
                   "</person>\n";  
regex reg("<(.\*)>(.\*)</(\\1)>");  
// print data with replacement for matched patterns  
cout << regex\_replace (data, // data  
                                  reg, // regular expression  
                                  "<$1 value=\"$2\"/>") // replacement  
       << endl;  
// same using sed syntax  
cout << regex\_replace (data, // data  
                                  reg, // regular expression  
                                  "<\\1 value=\"\\2\"/>", // replacement  
                                  regex\_constants::format\_sed) // format flag  
      << endl;  
// use iterator interface, and  
// - format\_no\_copy: don’t copy characters that don’t match  
// - format\_first\_only: replace only the first match found  
string res2;  
regex\_replace (back\_inserter(res2), // destination  
                      data.begin(), data.end(), // source range  
                      reg, // regular expression  
                      "<$1 value=\"$2\"/>", // replacement  
                      regex\_constants::format\_no\_copy | regex\_constants::format\_first\_only);// format flags  
cout << res2 << endl;  
  
                                            Regex Replacement Symbols

**9) Regex flags**  
  
Examples:  
// case-insensitive find LaTeX index entries  
string pat1 = R"(\\.\*index\{([^}]\*)\})"; // first capture group  
string pat2 = R"(\\.\*index\{(.\*)\}\{(.\*)\})"; // 2nd and 3rd capture group  
// grep and egrep can search for multiple regular expressions separatedby '\n' at the same time   
regex pat (pat1+"\n"+pat2, regex\_constants::egrep|regex\_constants::icase);  
// initialize string with characters from standard input:  
string data((istreambuf\_iterator<char>(cin)), istreambuf\_iterator<char>());  
// search and print matching index entries:  
smatch m;  
auto pos = data.cbegin();  
auto end = data.cend();  
for ( ; regex\_search (pos,end,m,pat); pos = m.suffix().first) {  
    cout << "match: " << m.str() << endl;  
    cout << " val: " << m.str(1)+m.str(2) << endl;  
    cout << " see: " << m.str(3) << endl;  
}  
  
Input samples:  
*\chapter{The Standard Template Library}  
\index{STL}%  
\MAININDEX{standard template library}%  
\SEEINDEX{standard template library}{STL}%  
This is the basic chapter about the STL.  
\section{STL Components}  
\hauptindex{STL, introduction}%  
The \stl{} is based on the cooperation of*  
...  
**10) The regex ECMAScript grammar and special chars**  
  
                                                                ECMAScript grammar  
  
                                                  Character classes in ECMAScript  
**(11)  Grammar differences**  
  
**(12) Regex exception**exception code to explanation string for debugging  
template <typename T>  
std::string regexCode (T code)  
{  
    switch (code) {  
        case std::regex\_constants::error\_collate:  
               return "error\_collate: regex has invalid collating element name";  
        case std::regex\_constants::error\_ctype:  
               return "error\_ctype: regex has invalid character class name";  
        case std::regex\_constants::error\_escape:  
               return "error\_escape: regex has invalid escaped char. or trailing escape";  
        case std::regex\_constants::error\_backref:  
               return "error\_backref: regex has invalid back reference";  
        case std::regex\_constants::error\_brack:  
               return "error\_brack: regex has mismatched ’[’ and ’]’";  
        case std::regex\_constants::error\_paren:  
               return "error\_paren: regex has mismatched ’(’ and ’)’";  
        case std::regex\_constants::error\_brace:  
               return "error\_brace: regex has mismatched ’{’ and ’}’";  
        case std::regex\_constants::error\_badbrace:  
               return "error\_badbrace: regex has invalid range in {} expression";  
        case std::regex\_constants::error\_range:  
               return "error\_range: regex has invalid character range, such as ’[b-a]’";  
        case std::regex\_constants::error\_space:  
               return "error\_space: insufficient memory to convert regex into finite state";  
        case std::regex\_constants::error\_badrepeat:  
               return "error\_badrepeat: one of \*?+{ not preceded by valid regex";  
        case std::regex\_constants::error\_complexity:  
               return "error\_complexity: complexity of match against regex over pre-set level";  
        case std::regex\_constants::error\_stack:  
               return "error\_stack: insufficient memory to determine regex match";  
    }  
    return "unknown/non-standard regex error code";  
}  
  
int main()  
{  
    try {  
        // initialize regular expression with invalid syntax:  
        regex pat ("\\\\.\*index\\{([^}]\*)\\}",  
        regex\_constants::grep|regex\_constants::icase);  
        ...  
    }  
    catch (const regex\_error& e) {  
        cerr << "regex\_error: \n"  
              << " what(): " << e.what() << "\n"  
              << " code(): " << regexCode(e.code()) << endl;  
    }  
}