Morden cpp notes — string

Monday, December 7, 2020 1:39 PM

string 其实并不是一个真正的类型，而是一个typedef：

```
using string = std::basic_string<char>; // string 其实是一个类型别名.
```

Eg.

```
string str = "abc";
assert(str.length() == 3);
assert(str < "xyz");
assert(str.substr(0,1) == "a");
assert(str[1] == 'b');
assert(str.find("1") = string::npos);
```

字符串和容器完全是两个不同的概念。字符串是文本，里面的字符是强关系，顺序不能随便调换，应视为一个整体处理。而容器是集合，里面的元素没有任何关系，可以随意增删改，更多地是操作单个元素。

如果确实需要存储字符的容器，比如字节序列，数据缓冲区，最好改用 vector<char>,

Tips

1.字面量后缀.

C++11 新增了后缀"s"，明确是 string 字符串，不是C字符串，可以利用auto作类型推导.

```
using namespace std::literals::string_literals; //必须打开名字空间.
auto str = "std string"s; //后缀s,表示是标准字符串.直接类型推导.
```

2.原始字符串. Raw string literal.

```
auto str = R"(nier:automata)"; //原始字符串 nier:automata.
```

C++字符串有转义用法，在字符前加"\"，就可以用"\n"，"\t"表示回车，跳格等不可打印字符.

但有时我们只想用字符串的原始形式，e.g.

```
auto str1 = R"(char""'')"; // 输出 char""''
auto str2 = R"(\r\n\t\")"; // 输出 \r\n\t\"
auto str4 = "\\\\\\$"; //转义后输出 \\\$.
auto str5 = R"==(R"(xxx)")=="; // 输出 R"(xxx)".
```

3.字符串转换函数.

```
assert(stoi("42") == 42); //字符串转整数
assert(stol("253") == 253L); //字符串转长整数
assert(stod("2.0") == 2.0); //字符串转浮点数.
assert(to_string(1984) == "1984"); //整数转字符串.
```

4.字符串视图类.

大字符串的拷贝，修改代价很高，所以尽量用 const string&.

在C++17里，string_view是一个字符串的视图，成本很低，拷贝和修改都非常廉价.

可以在C++11里，实现一个简单的版本：

```
class my_string_view final //字符串视图类,示范实现.
{
public:
    using this_type = my_string_view; //各种内部类型定义.
    using string_type = std::string;
    using string_ref_type = const std::string&;

    using char_ptr_type = const char*;
    using size_type = size_t;

private:
    char_ptr_type ptr = nullptr; //字符串指针.
    size_type len = 0; //字符串长度.

public:
    my_string_view() = default;
   ~my_string_view() = default;

    my_string_view(string_ref_type str) noexcept
        : ptr(str.data()), len(str.length())
    {}

public:
   char_ptr_type data() const //常函数,返回字符串指针.
   {
     return ptr;
   }

   size_type size() const //常函数,返回字符串长度.
   {
     return len;
   }
};
```

正则表达式

- regex_match() 完全匹配一个字符串.
- regex_search() 在字符串里查找一个正则匹配
- regex_replace() 正则查找再替换.

记得使用原始字符串.

```
auto make_regex = [](const auto& txt) //生产正则表达式.
{
  return std::regex(txt);
};
auto make_match = []() //生产正则匹配结果
{
  return std::smatch();
};
auto str = "neir:automata"s; //待匹配的字符串.
auto reg = make_regex(R"(^(\w+)\:(\w+)$)"); //原始字符串定义正则表达式.
auto what = make_match();
assert(regex_match(str, what, reg));
for(const auto& x: what) {
    cout << x << ",";
}
```

先定义了两个简单的 lambda表达式，生产正则对象，主要为了方便使用auto自动类型推导，同时也隐藏了具体的类型信息，将来可以随时变化，然后调用 regex_match() 检查字符串，函数返回bool表示是否完全匹配正则，如果匹配成功，结果存储在 what 里，可以像容器那样访问，第0号元素是整个匹配串，其他的是子表达式匹配串.

regex_search(), regex_replace() 用法也差不多.

```
auto str = "god of war"s; //待匹配字符串.
auto reg = make_regex(R"((\w+)\s(\w+))"); //原始字符串定义正则表达式.
auto what = make_match(); //准备获取结果.
auto found = regex_search(str, what, reg); //正则查找.
assert(found); //断言找到匹配.
assert(!what.empty()); //断言有匹配结果.
assert(what[1] == "god"); //看第一个子表达式.
assert(what[2] == "of"); //看第二个子表达式.
auto new_str = regex_replace(      //正则替换,返回新字符串
    str,                           //原字符串不改动
    make_regex(R"(\w+$)"),         //就地生成正则表达式对象.
    "peace"                        //需要指定替换的文字.
  );
cout << new_str << endl; //输出 god of peace.
```

Tips:

- 在string 转换C字符串的时候，注意 c_str()和data()的区别，两个函数都返回 const char* 指针，但c_str()会在末尾添加一个'\0'。