Анализатор зависимостей файлов С++

1. Постановка задачи

Нужно написать анализатор файлов исходного кода на С++. Программа должна выводить дерево зависимостей и частоты включений файлов. Если найдены включения несуществующих файлов, то программа должна вывести пометку «(!)» после имени файла.

В директории с исходными кодами могут быть вложенные папки (то есть обход должен производиться рекурсивно). Файлы имеют расширения ".hpp" и ".cpp"

Требования к выполнению задачи:

· Задача выполняется на С++ (стандарт C++11, C++14 или C++17 на выбор)

· Использование стандартной библиотеки C++ Standard Library, разрешается использование библиотеки Boost

· Использование сборочной системы CMake.

· Программа должна являться консольным приложением

· Программа должна собираться под Windows при помощи компилятора Clang либо MSVC и под Linux – с помощью компилятора Clang либо GCC.

· Решение должно состоять из документа, описывающего реализацию, исходных кодов программы и исполняемого модуля, демонстрирующего работу класса.

Входные параметры

В качестве аргументов командной строки получает путь к каталогу с исходниками и пути для поиска заголовочных файлов (то есть файлы, включенные директивой вида #include <a.hpp> должны искаться в каталогах, указанных в параметрах командной строки)

analyzer <sources path> [options]

Опции:

-I – путь для поиска исходных файлов

Пример вызова программы:

analyzer d:\mysources\ -I d:\mysources\includes -I d:\mylibrary

(программа должна проанализировать исходники в каталоге d:\mysources, используя следующие пути для поиска заголовочных файлов: "d:\mysources\include" и "d\mylibrary")

Вывод результата

Программа должна выводить дерево зависимостей исходных файлов и список файлов с частотами их включения в другие файлы.

Список частот включений файлов должен быть отсортирован в порядке убывания частоты; записи с одинаковым значением частоты сортируются по алфавиту. Список частот отделён от дерева зависимостей пустой строкой.

Пример вывода:

test.cpp

..a.hpp

....foo.hpp

....b.hpp

......foo.hpp

..test.hpp

....foo.hpp

..deleted.hpp (!)

main.cpp

..test.hpp

....foo.hpp

foo.hpp 4

test.hpp 2

a.hpp 1

b.hpp 1

deleted.hpp 1

main.cpp 0

test.cpp 0

Примечания

1. Необходимо учитывать, что директивы #include могут быть закомментированы

2. Утилита должна работать в соответствии с правилами работы препроцессора С++: если имя файла заключено в кавычки, то поиск производится относительно пути к обрабатываемому файлу; если оно заключено в "< >", то файл ищется в путях, указанных в агрументах при запуске программы

#include "foo.hpp"

#include <test/aa.hpp>

/\*

#include <c.hpp>

\*/

В данном примере включены файлы foo.hpp и test/aa.hpp; файл c.hpp не включен. Поиск файла foo.hpp производится в текущей директории, а файла test/aa.hpp - в директориях, заданных в аргументах командной строки.

3. Желательно предусмотреть обработку циклических зависимостей

1. Описание работы алгоритма программы

Изначально нужно правильно найти и прочитать присутствующие файлы с расширением .cpp в директории исходных файлов (argv[1]). Для глубокого исследования директории, была использована библиотека boost::filesystem. С помощью рекурсивного итератора был сформирован контейнер, хранящий адреса исходных файлов. Решение описано функцией directory\_read()

Далее для каждого файла по отдельности вызывается функция finclude(). В неё передаётся путь каждого найденного файла с расширением .cpp, файл открывается и каждый инклуд, который есть внутри, записывается без кавычек в отдельный контейнер. Каждую из записей контейнера после мы помещаем в функцию includeexists(), чтобы найти физическое расположение файлов в каталогах инклудов (argv[n]).

В функции includeexists() с помощью рекурсивного итератора boost::filesystem мы находим все адреса существующих физически инклудов, записываем их в вектор. Параллельно считаем повторения одинаковых включений файлов, формируя вектор структур и ловим названия не найденных физически инклудов в каталоге при поиске.

Возвращаемся в функцию finclude. Если название инклуда не равно какому-нибудь из названий в контейнере, открываем этот инклуд фукнцией includeexists() рекурсивно. Если у файла больше нет инклудов, формируем строку way\_adr из элементов контейнера, которая выглядит так:   
файл\_0 >> файл\_1 >> файл\_2 >> файл\_3. Получаем набор из таких строк.

Дальше группируем одинаковые названия функцией result\_out(). Например:  
Файл\_0 >> файл\_1 >> файл\_2   
>> файл\_5  
>> >> файл\_15  
>> >> >> файл\_22 >> файл\_23  
>> файл\_35

В этой же функции выводим перехваченные имена инклудов, отсутствующих в каталоге физически.

В main выводим список инклудов и частоту их включений.

1. Требования и руководство к эксплуатации программы

Для компиляции исходного кода должен должна быть подключена библиотека boost\_filesystem-vc142. Для запуска анализатора используется командная строка, в которую записывается название программы и директории для поиска. Ограничений на количество директорий нет инклудов, проверял я на двух.

Пример:

analyze \dir\ \dir\ \includes\

1. Исходный код программы:
2. #include <iostream>
3. #include <string>
4. #include <vector>
5. #include <fstream>
6. #include <boost/filesystem.hpp>
7. using std::cout;
8. using std::endl;
9. using std::string;
10. using namespace boost::filesystem;
11. struct repeat
12. {
13. string name;
14. int c;
15. };
16. std::vector<repeat> r;
17. repeat ob;
18. int directory\_read(char\* str);
19. int finclude(const char\* str1);
20. int includeexists(const char\* str2, std::vector<string> &t);
21. int result\_out(void);
22. std::vector<path> programfiles; //адреса .cpp файлов
23. std::vector<string> pfiles; //Кастование из path в string
24. std::vector<string> way\_adr; //адреса путей
25. std::vector<string> flist; //Список инклюдов
26. std::vector<string> argv\_dir; //Все адреса argv
27. std::vector<path> includefiles; //Сюда адреса инклюдов из директории
28. std::vector<std::vector<string>> container\_containerov;
29. std::vector<string> disabled; //Не найдены в каталоге
30. std::vector<path> programnames; //Сюда адреса исходников
31. void show\_info()
32. {
33. cout << endl << "Для поиска введите: analyze ... - имя каталога -I ...\\... - адрес поиска 1 -I \\... - адрес поиска 2" << endl;
34. }
35. int main(int argc, char\* argv[])
36. {
37. setlocale(LC\_ALL, "Russian");
38. if (argc > 2)
39. {
40. for (int i = 2; i < argc; i++)
41. argv\_dir.push\_back(argv[i]);
42. r.push\_back(ob);
43. cout << "\nАнализ директории исходников\n";
44. directory\_read(argv[1]);
45. r.pop\_back();
46. for (int i = 0; i < r.size(); i++)
47. cout << r.at(i).name << " " << r.at(i).c << endl;
48. }
49. else
50. {
51. cout << "Что-то пошло не так... Внимательно изучите инструкцию." << endl;
52. show\_info();
53. return 0;
54. }
55. return 0;
56. }
57. int directory\_read(char\* str0) {
58. path p(str0);
59. try
60. {
61. if (exists(p))
62. {
63. if (is\_directory(p))
64. {
65. for (recursive\_directory\_iterator iter(p), end; iter != end; ++iter) // Перебор файлов в папке
66. {
67. if (!is\_directory(iter->path()) && iter->path().extension() == ".cpp") // Поиск совпадений по расширению
68. programfiles.push\_back(iter->path());
69. }
70. for (auto&& x : programfiles)
71. programnames.push\_back(x.filename());
72. std::sort(programfiles.begin(), programfiles.end());
73. std::vector<path>::iterator ite = programfiles.begin();
74. while (ite != programfiles.end())
75. {
76. pfiles.push\_back((\*ite).string());
77. ++ite;
78. }
79. for (int i = 0; i < pfiles.size(); i++)
80. {
81. finclude((pfiles[i]).c\_str());
82. result\_out();
83. container\_containerov.clear();
84. }
85. }
86. else
87. cout << p << " Существует, но не является каталогом\n";
88. }
89. else
90. cout << p << " Не существует\n";
91. }
92. catch (const filesystem\_error& ex)
93. {
94. cout << ex.what() << '\n';
95. }
96. return 0;
97. }
98. int finclude(const char\* str1)
99. {
100. way\_adr.push\_back(str1);
101. path pa(str1);
102. string line; //переменная для чтения строки
103. string str\_comment = "//";
104. string bl\_comment\_s = "/\*";
105. string bl\_comment\_f = "\*/";
106. string include = "#include";
107. bool comment = false;
108. ifstream in(pa); //файл для чтения
109. flist.clear();
110. if (in.is\_open())
111. {
112. while (getline(in, line)) //читаем по строкам
113. {
114. if (line.find(bl\_comment\_s, 0) != string::npos) // Если начался блок коментариев
115. {
116. if (line.find(bl\_comment\_f, 0) != string::npos) //Если он закончился в этой же строке.
117. {
118. string start = line.substr(0, line.find(bl\_comment\_s, 0));
119. string end = line.substr((int)line.find(bl\_comment\_f) + bl\_comment\_f.length());
120. line = start + end; // В темп пишем всё, что между этими двумя строками
121. }
122. else
123. {
124. comment = true; // Если комментарий только начался в этой строке, а закончится после.
125. }
126. }
127. if (comment && (line.find(bl\_comment\_f, 0) != string::npos)) // Если нашли конец незакрытого блока комметария.
128. {
129. line = line.substr((int)line.find(bl\_comment\_f) + bl\_comment\_f.length());
130. comment = false;
131. }
132. if (comment)
133. {
134. continue; //Пока мы в блоке коммента, дальше можно не выполнять
135. }
136. if (line.find(str\_comment, 0) != string::npos)
137. {
138. line = line.substr(0, (int)line.find(str\_comment, 0));
139. // Уничтожаем все данные, после линейного комментария.
140. }
141. if (line.find(include, 0) == string::npos)
142. {
143. continue; //в строке нет include
144. }
145. string answer;
146. answer = line.substr(10);
147. flist.push\_back(answer.erase(answer.size() - 1)); //имена инклюдов
148. }
149. }
150. in.close();
151. std::vector<string> temp;
152. for (int i = 0; i < flist.size(); i++)
153. includeexists(flist[i].c\_str(), temp);
154. bool flag\_overlap=false;
155. if (temp.size() > 0)
156. {
157. for (int j = 0; j < temp.size(); j++)
158. {
159. for (int i = 0; i < way\_adr.size(); i++)
160. if (temp[j] == way\_adr[i])
161. {
162. flag\_overlap = true;
163. break;
164. }
165. if (!flag\_overlap)
166. {
167. finclude(temp[j].c\_str());
168. }
169. else flag\_overlap = false;
170. if (j == temp.size() - 1)
171. way\_adr.pop\_back();
172. }
173. }
174. else
175. {
176. container\_containerov.push\_back(way\_adr);
177. way\_adr.pop\_back();
178. }
179. return 0;
180. }
181. int includeexists(const char\* str2, std::vector<string> &t)
182. {
183. bool fl = false;
184. for (int i = 0; i < r.size(); i++)
185. {
186. if (r.at(i).name == str2)
187. {
188. r.at(i).c++;
189. fl = false;
190. break;
191. }
192. else fl = true;
193. }
194. if (fl)
195. {
196. r.at(r.size()-1).name = str2;
197. r.at(r.size()-1).c = 1;
198. r.push\_back(ob);
199. }
200. bool flag\_empty = false;
201. includefiles.clear();
202. for (int i = 0; i < argv\_dir.size(); i++)
203. {
204. path p(argv\_dir[i]);
205. try
206. {
207. if (exists(p))
208. {
209. if (is\_directory(p))
210. {
211. for (recursive\_directory\_iterator it(p), end; it != end; ++it) // Перебор файлов в папке
212. {
213. if (!is\_directory(it->path()) && it->path().string() == (argv\_dir[i] + str2))
214. { // Поиск совпадений по str2
215. includefiles.push\_back(it->path());
216. flag\_empty = false;
217. break;
218. }
219. else
220. flag\_empty = true;
221. }
222. if (!flag\_empty) break;
223. }
224. else
225. cout << p << " Существует, но не является файлом или каталогом\n";
226. }
227. else
228. cout << p << " Не существует\n";
229. }
230. catch (const filesystem\_error& ex)
231. {
232. cout << ex.what() << '\n';
233. }
234. }
235. if (flag\_empty)
236. disabled.push\_back(str2);
237. for (int i = 0; i < includefiles.size(); i++)
238. t.push\_back(includefiles[i].string());
239. return 0;
240. }
241. int result\_out(void)
242. {
243. for (int i = 0; i < container\_containerov[0].size(); i++)
244. {
245. for (int j = 0; j < i; j++)
246. cout << "..";
247. cout << container\_containerov[0][i] << endl;
248. }
249. for (int k = 1; k < container\_containerov.size(); k++)
250. {
251. for (int i = 0; i < container\_containerov[k].size(); i++)
252. {
253. if ((container\_containerov[k - 1].size() > i) && (container\_containerov[k - 1][i] == container\_containerov[k][i]))
254. cout << "";
255. else {
256. for (int j = 0; j < i; j++)
257. cout << "..";
258. cout << container\_containerov[k][i] << endl;
259. }
260. }
261. }
262. for (int z = 0; z < disabled.size(); z++)
263. cout << disabled[z] << " (!) " << endl;
264. disabled.clear();
265. return 0;
266. }

Приложение 1

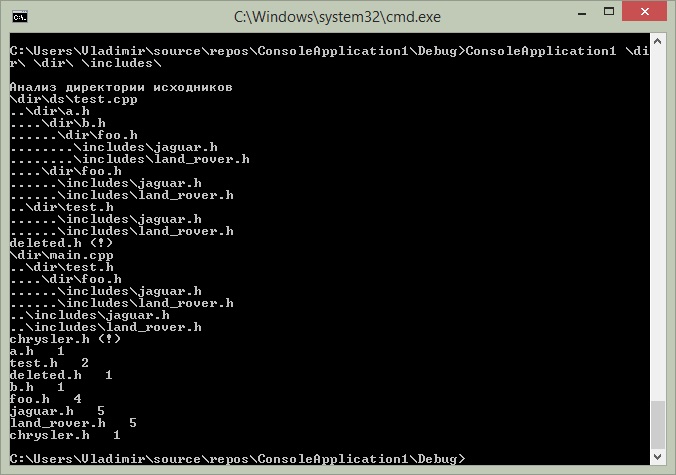


Рисунок 1 – Результат выполнения программы.