LP Assignment 2

BT19CSE**122** - Deep Walke

01-04-2022

Language: **C++**

PROBLEM STATEMENT

The assignment is to demonstrate the usage of any static analysis and any dynamic analysis tool.

Static and Dynamic Analysis is tested on the code of an LALR Parser written in C++.
Code Link

STATIC ANALYSIS

Tool Used: *cppcheck*

Static code analysis is a technique for debugging and detecting problems in source code before a program is run. It's done by comparing a list of codes against a list of coding rules.

- ★ **Advantages**
- ➔ Locating troublesome code snippets at their exact position.
- ➔ Error detection in code
- ➔ Undefined behavior
- ➔ Using dangerous code patterns
- ➔ Because code is often run more than it is compiled, detecting code that might be improved for better performance.
- ➔ Coding style

Installing cppcheck

```
root@LAPTOP-1K9986MA:/mnt/c/windows/system32# sudo apt install cppcheck
Reading package lists... Done
Building dependency tree
Reading state information... Done
cppcheck is already the newest version (1.82-1).
0 upgraded, 0 newly installed, 0 to remove and 182 not upgraded.
root@LAPTOP-1K9986MA:/mnt/c/windows/system32#
```

CPPCHECK

Static analysis is performed with the Cppcheck utility (Cppcheck). Cppcheck is a C/C++ code static analysis tool. It is used to examine code by finding any errors or troublesome sections of code, as well as attempting to minimize false positives.

For analysis:

- Run static analysis on c++ code using the following command
  - `cppcheck --enable=all --suppress=missingIncludeSystem lalr.cpp`

The types of severities in the output of analysis are :

- **error**: when there is an undefined behavior during the execution of the code, such as a memory leak or resource leak
- **warning**: when code is executed there might be undefined behavior
- **style**: stylistic problems, such as redundant code, unused functions, etc
- **performance**: run time performance optimization
- **portability**: portability warnings like 64-bit portability, etc
- **information**: configuration problems

Output Screenshots & Explanation

```
root@LAPTOP-1K9986MA:/mnt/c/Deep/6th sem/LP/Assignment3# cppcheck --enable=all --suppress=
missingIncludeSystem lalr.cpp>err.txt
[lalr.cpp:705]: (style) The scope of the variable 'i' can be reduced.
[lalr.cpp:297]: (style) Unused variable: itm
[lalr.cpp:705]: (style) Unused variable: j
[lalr.cpp:78]: (error) Iterators of different containers are used together.
[lalr.cpp:87]: (error) Iterators of different containers are used together.
[lalr.cpp:142]: (error) Iterators of different containers are used together.
[lalr.cpp:160]: (error) Iterators of different containers are used together.
root@LAPTOP-1K9986MA:/mnt/c/Deep/6th sem/LP/Assignment3#
```

For the code, we have the following severities

- variableScope **(style)**
  - The output says that the scope of variable **i** can be changed to make the code efficient.

```
705     int i, j;
```

- Unused Variable **(style)**
  - This is raised when a variable is defined but not used anywhere.
  - Here we got **itm** and **j** as unused variables.

```
297     item itm;
```

```
705     int i, j;
```

➢ Iterators of different containers are used together. **(error)**

```
142                 if (flag && find(begin(nonterminals), end(nonterminals), p) != nonterminals.end())
```

- Here in the above screenshot, we can see we are using two different iterators together.
- The similar error we got on the **line 78, line 87, line 142, and line 160** of the code.

DYNAMIC ANALYSIS
Tool Used: *gprof*

Profiling

- ★ **Flat Profile:** It shows how much time the program spent on each function, and the number of times that function was called.
- ★ **Call Graph:** It shows for each function, which functions called it and which other functions it called, and how many times. There is an estimate of how much time was spent in the subroutines of each function.

➢ **Gprof**
  - It gives us an execution profile of our C/C++ program.
  - It basically calculates the amount of time spent in each routine or function.

➢ Now we will use the Gprof tool.

➢ First, we need to check whether the gprof tool is installed or not in the system using the command **gprof.**

```
root@LAPTOP-1K9986MA:/mnt/c/Deep/6th Sem/LP/Assignment3# gprof
a.out: No such file or directory
```

This shows that the tool is **already installed.**

➢ we have compiled the code using g++

➢ To profile the code using Gprof, we have to use the **-pg** command-line option. So, the command becomes
g++ -pg lalr.cpp -o lalr_gprof

➢ Run the executable file:
./lalr_gprof

```
root@LAPTOP-1K9986MA:/mnt/c/Deep/6th Sem/LP/Assignment3# g++ -pg lalr.cpp -o lalr_gprof
root@LAPTOP-1K9986MA:/mnt/c/Deep/6th Sem/LP/Assignment3# ./lalr_gprof
Terminals:
 )
Non Terminals

Productions
 | e | )
First Set
Follow Set
 -> $
```

➢ After compiling and running the code, a **gmon.out** got generated.

```
root@LAPTOP-1K9986MA:/mnt/c/Deep/6th Sem/LP/Assignment3# ls
err.txt  gmon.out  input.txt  lalr.cpp  lalr_gprof  output.txt
```

➢ gmon.out file will contain all the information that the Gprof tool requires to produce human-readable profiling data.

➢ Now use the gprof tool for analysis with the command:
gprof lalr_gprof gmon.out > output.txt

```
root@LAPTOP-1K9986MA:/mnt/c/Deep/6th sem/LP/Assignment3# gprof lalr_gprof gmon.out > output.txt
```

➢ We can see the output in the **output.txt** file.

Output Screenshots & Explanation

➢ The output is divided into two parts
- Flat Profile
- Call Graph

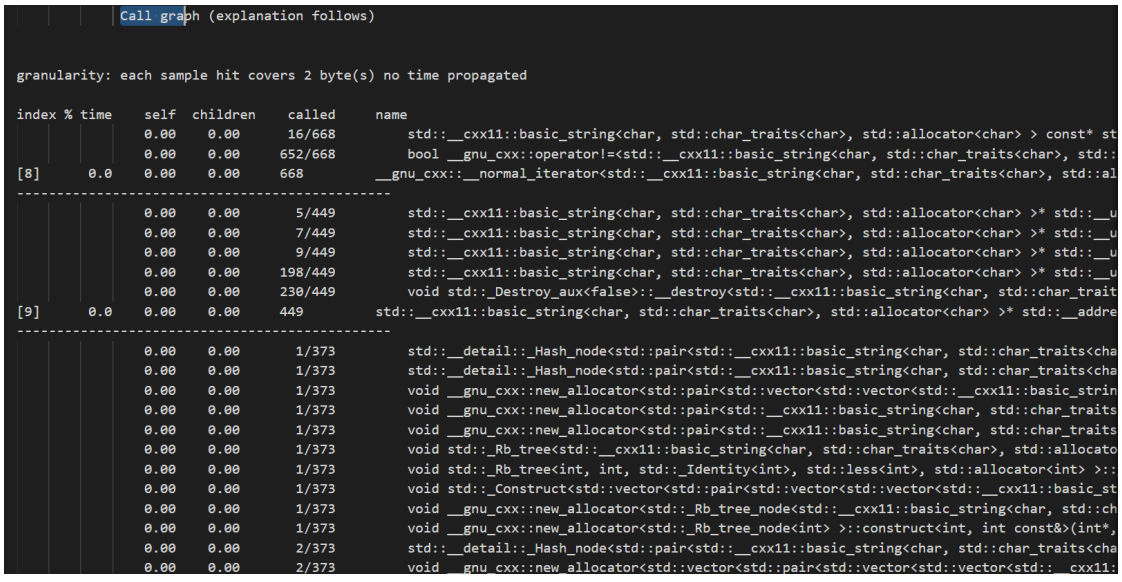
★ **Flat Profile:** It shows how much time the program spent on each function, and the number of times that function was called.

```
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00      668     0.00     0.00  __gnu_cxx::__normal_iterator<std::__cxx11::basic_string<char, std::char_traits<char>
  0.00      0.00     0.00      449     0.00     0.00  std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >* std
  0.00      0.00     0.00      373     0.00     0.00  operator new(unsigned long, void*)
  0.00      0.00     0.00      350     0.00     0.00  std::_Vector_base<std::__cxx11::basic_string<char, std::char_traits<char>, std::allo
  0.00      0.00     0.00      326     0.00     0.00  bool __gnu_cxx::operator!=<std::__cxx11::basic_string<char, std::char_traits<char>,
  0.00      0.00     0.00      298     0.00     0.00  __gnu_cxx::new_allocator<std::__cxx11::basic_string<char, std::char_traits<char>, st
  0.00      0.00     0.00      298     0.00     0.00  std::allocator<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocat
  0.00      0.00     0.00      281     0.00     0.00  __gnu_cxx::new_allocator<std::__cxx11::basic_string<char, std::char_traits<char>, st
  0.00      0.00     0.00      281     0.00     0.00  std::allocator<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocat
  0.00      0.00     0.00      272     0.00     0.00  __gnu_cxx::__normal_iterator<std::__cxx11::basic_string<char, std::char_traits<char>
  0.00      0.00     0.00      256     0.00     0.00  bool __gnu_cxx::__is_null_pointer<char>(char*)
  0.00      0.00     0.00      256     0.00     0.00  void std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >
  0.00      0.00     0.00      256     0.00     0.00  std::iterator_traits<char*>::difference_type std::__distance<char*>(char*, char*, st
  0.00      0.00     0.00      256     0.00     0.00  std::iterator_traits<char*>::iterator_category std::__iterator_category<char*>(char*
  0.00      0.00     0.00      256     0.00     0.00  std::iterator_traits<char*>::difference_type std::distance<char*>(char*, char*)
  0.00      0.00     0.00      242     0.00     0.00  __gnu_cxx::__normal_iterator<std::vector<std::__cxx11::basic_string<char, std::char_
  0.00      0.00     0.00      242     0.00     0.00  std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > cons
  0.00      0.00     0.00      230     0.00     0.00  void std::_Destroy<std::__cxx11::basic_string<char, std::char_traits<char>, std::all
  0.00      0.00     0.00      214     0.00     0.00  std::vector<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<
  0.00      0.00     0.00      214     0.00     0.00  void std::_Construct<std::__cxx11::basic_string<char, std::char_traits<char>, std::a
  0.00      0.00     0.00      198     0.00     0.00  __gnu_cxx::__normal_iterator<std::__cxx11::basic_string<char, std::char_traits<char>
  0.00      0.00     0.00      198     0.00     0.00  __gnu_cxx::__normal_iterator<std::__cxx11::basic_string<char, std::char_traits<char>
```

★ Meaning of each field that is defined in **Flat Profile.**

%time	the percentage of the total running time of the program used by this function.
cumulative seconds	a running sum of the number of seconds accounted for by this function and those listed above it.
self seconds	the number of seconds accounted for by this function alone. This is the major sort for this listing.
calls	the number of times this function was invoked, if this function is profiled, else blank.
self ms/call	the average number of milliseconds spent in this function per call, if this function is profiled else blank
total ms/call	The average number of milliseconds spent in this function and its descendants per call, if this function is profiled, else blank.
name	Name of the function

★ **Call Graph:** It shows for each function, which functions called it and which other functions it called, and how many times. There is an estimate of how much time was spent in the subroutines of each function.



```
Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) no time propagated

index % time    self  children    called     name
                0.00    0.00      16/668         std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const* st
                0.00    0.00     652/668         bool __gnu_cxx::operator!=<std::__cxx11::basic_string<char, std::char_traits<char>, std::
[8]     0.0     0.00    0.00     668         __gnu_cxx::__normal_iterator<std::__cxx11::basic_string<char, std::char_traits<char>, std::al
-----------------------------------------------
                0.00    0.00       5/449         std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >* std::__u
                0.00    0.00       7/449         std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >* std::__u
                0.00    0.00       9/449         std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >* std::__u
                0.00    0.00     198/449         std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >* std::__u
                0.00    0.00     230/449         void std::_Destroy_aux<false>::__destroy<std::__cxx11::basic_string<char, std::char_trait
[9]     0.0     0.00    0.00     449         std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >* std::__addre
-----------------------------------------------
                0.00    0.00       1/373         std::__detail::_Hash_node<std::pair<std::__cxx11::basic_string<char, std::char_traits<cha
                0.00    0.00       1/373         std::__detail::_Hash_node<std::pair<std::__cxx11::basic_string<char, std::char_traits<cha
                0.00    0.00       1/373         void __gnu_cxx::new_allocator<std::pair<std::vector<std::vector<std::__cxx11::basic_strin
                0.00    0.00       1/373         void __gnu_cxx::new_allocator<std::pair<std::__cxx11::basic_string<char, std::char_traits
                0.00    0.00       1/373         void __gnu_cxx::new_allocator<std::pair<std::__cxx11::basic_string<char, std::char_traits
                0.00    0.00       1/373         void std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocato
                0.00    0.00       1/373         void std::_Rb_tree<int, int, std::_Identity<int>, std::less<int>, std::allocator<int> >::
                0.00    0.00       1/373         void std::_Construct<std::vector<std::pair<std::vector<std::vector<std::__cxx11::basic_st
                0.00    0.00       1/373         void __gnu_cxx::new_allocator<std::_Rb_tree_node<std::__cxx11::basic_string<char, std::ch
                0.00    0.00       1/373         void __gnu_cxx::new_allocator<std::_Rb_tree_node<int> >::construct<int, int const&>(int*,
                0.00    0.00       2/373         std::__detail::_Hash_node<std::pair<std::__cxx11::basic_string<char, std::char_traits<cha
                0.00    0.00       2/373         void __gnu_cxx::new_allocator<std::vector<std::pair<std::vector<std::vector<std::__cxx11:
```

★ Below three screenshots are to explain the information the call graph contains:

❖ Each entry in this table consists of several lines. The line with the index number at the left-hand margin lists the current function. The lines above it list the functions that called this function, and the lines below it list the functions this one called.

index	A unique number that is given to each element of the table.
%time	the percentage of the **total** time that was spent in this function and its children.
self	This is the total amount of time spent in this function.
children	This is the total amount of time propagated into this function by its children.
called	the number of times this function was called.
name	Name of the function

❖ For the function's parents, the fields have the following meanings:

self	This is the total amount of time that was propagated directly from the function into this parent.
children	This is the total amount of time propagated from the function's children into this parent.
called	the number of times this parent called the function `/' the total number of times the function was called.
name	Name of the parent

- If the parents of the function cannot be determined, the word `<spontaneous>' is printed in the `name' field, and all the other fields are blank.

self	This is the amount of time that was propagated directly from the child into the function.
children	This is the amount of time that was propagated from the child's children to the function.
called	This is the number of times the function called this child `/' the total number of times the child was called
name	Name of the child
