Реквизиты

Жолобов Даниил Валерьевич
3 курс
z33434
2024

ЛР #5: [C++ & UNIX]: C++ OOP / PARALLEL

Цель

Познакомить студента с принципами объектно-ориентированного программирования на примере создания сложной синтаксической структуры. Придумать синтаксис своего персонального мини-языка параллельного программирования, а также реализовать его разбор и вычисление.

Задача

1. [C++ PARALLEL LANG] Создать параллельный язык программирования

   > Требуется создать язык программирования, в котором будет доступна установка следующих команд:

   - Установка счетного цикла
   - Вывод в консоль
   - Вывод в файл в режиме добавления
   - Арифметические операции +, -, *, /

Счетный цикл должен поддерживать дальнейшую установку всех остальных поддерживаемых команд.

Для реализации задачи использовать технологию объектно-ориентированного программирования в части реализации поддерживаемых команд языка

В программе должны быть отражены следующие шаги:

1. Текстовый ввод команд. Каждая новая строка – это новый набор команд.
2. Ожидание команды на окончание ввода
3. Параллельное исполнение введенных строк (наборов команд). Наборы команд должны исполняться параллельно. В консоли фиксировать время запуска / завершения каждого потока. При выводе информации о времени указывать принадлежность потока к строке (набору команд)
4. [LOG] Результат всех вышеперечисленных шагов сохранить в репозиторий (+ отчет по данной ЛР в папку doc)

> Фиксацию ревизий производить строго через ветку dev. С помощью скриптов накатить ревизии на stg и на prd. Скрипты разместить в корне репозитория. Также создать скрипты по возврату к виду текущей ревизии (даже если в папке имеются несохраненные изменения + новые файлы).

Решение

1. [C++ PARALLEL LANG] Создать параллельный язык программирования

   > Требуется создать язык программирования, в котором будет доступна установка следующих команд:

   - Установка счетного цикла
   - Вывод в консоль
   - Вывод в файл в режиме добавления
   - Арифметические операции +, -, *, /

Счетный цикл должен поддерживать дальнейшую установку всех остальных поддерживаемых команд.

Для реализации задачи использовать технологию объектно-ориентированного программирования в части реализации поддерживаемых команд языка

В программе должны быть отражены следующие шаги:
1.1. Текстовый ввод команд. Каждая новая строка – это новый набор команд.

Поскольку в каждую строку можно уместить несколько команд, потребуется использовать разделитель команд. Пусть им будет ';'. Поэтому будет разумно выбрать инструкции, похожие на имеющиеся в C++, но упростить их. Язык будет похож на cpp и параллельным, так что можно его назвать "ppp" - parallel cpp.

1.2. Ожидание команды на окончание ввода

Введём для этого отдельную инструкцию:
run_ppp;

1.3. Параллельное исполнение введенных строк (наборов команд). Наборы команд должны исполняться параллельно. В консоли фиксировать время запуска / завершения каждого потока. При выводе информации о времени указывать принадлежность потока к строке (набору команд)

Распараллеливать ход выполнения может иметь смысл в некоторых циклах и некоторых последовательных наборах команд. Чтобы понять, в каких случаях следует выполнять параллельные вычисления, можно построить граф изменения переменных.

Для начала рассмотрим простой случай - только последовательность команд, без циклов. Тогда достаточно проанализировать следующие инструкции:
- variable definition [T] (создание параллельной переменной

из обычной)
- variable redefinition [T,T] (присваивание параллельной переменной другой параллельной переменной)
- operators[T,T] -> T (операторы арифметических операций)
- printf to std_out (вывод в консоль)
- fprintf to to file (вывод в файл)

Попробуем реализовать все операции в контексте класса ppp
- variable definition [T] - ppp.add_variable(T)->expression(T);
- variable redefinition [T,T] - p.expression_assignment(T, T);
- operators[T,T] -> T - как перегрузку операторов для класса expression
- printf to std_out - p.add_printer() << expression
- fprintf to to file - p.add_fprinter(filename) << expression

Поскольку необходимо поддерживать по крайней мере базовые числовые типы переменных, реализуем управление операциями через базовый абстрактный класс base для класса expression.

Далее можно реализовать простой компилятор для .ppp файлов, который будет оборачивать их в структуру минимального .cpp файла, подключая header с классом ppp и компилирующий при помощи clang++ получившийся .cpp файл. Следовательно, удобно выбрать такой синтаксис языка, который легко реализовать при помощи #define.

```
#define start_ppp ppp p;
#define ppp_show_info p.print_thread_info = true;
#define run_ppp p.calculate();
#define var(v) p.add_variable((v));
#define ppp_assign(from, to)
p.expression_assignment((from), (to));
#define start_pfor(varname, a, b, c) \
```

```
        p.for_loop(a, b, c,function([&](ppp& p, decltype(a)
varname) {
#define end_pfor \
        }));
#define pcout p.add_printer()
#define pfout(filename) p.add_fprinter((filename))
```

Выполнить тесты:

```
clang++ -Wall -Wextra -Weffc++ -O3 lab_05/src/1_ppp_tests.cpp -o
lab_05/build/a.out
lab_05/build/a.out
```

```
clang++ -Wall -Wextra -Weffc++ -O3 lab_05/src/3_ppp_compiler.cpp -o
lab_05/src/comp
lab_05/src/comp lab_05/src/fibo.ppp
lab_05/src/fibo.exe
```

1. [LOG] Результат всех вышеперечисленных шагов сохранить в репозиторий (+ отчет по данной ЛР в папку doc)

```
Фиксацию ревизий производить строго через ветку dev. С
помощью скриптов накатить ревизии на stg и на prd. Скрипты
разместить в корне репозитория. Также создать скрипты по
возврату к виду текущей ревизии (даже если в папке имеются
несохраненные изменения + новые файлы).
```

Заключение

Получилось реализовать примитивный параллельный язык программирования - ppp. Поддерживаются несколько инструкций, произвольные типы и простые бинарные операторы для них. Для языка создан синтаксис на основе `#define` и простой компилятор, который выполняет создание бинарного файла из .ppp файла через промежуточный .cpp файл.

Выполнив ЛР, я закрепил тему абстрактных классов, узнал про правила пяти и трёх, ознакомился со способами перегрузки бинарных операторов, научился работать с std::async. Кроме того, изучил тему copy elision, RVO|NRVO и пришёл к выводу, что размещать логику в конструкторах не следует. Приобрёл некоторый опыт работы с указателями в дебаггере VS Code и настройке clang-format.