Лабораторная работа № 6

Средства автоматизации разработки программ

Цель работы

Отработать применение средств автоматизации разработки программ:

1) статических анализаторов исходного кода на примере CppCheck;
2) генераторов документации на примере Doxygen;
3) анализаторов производительности на примере GNU Profiler;
4) систем сборки на примере Make и CMake.

Подготовка к лабораторной работе

1. Скопировать в отдельный каталог собственные результаты выполнения последней из завершенных лабораторных работ. Например, для ЛР № 5, нужно скопировать `LinkedList.h` и `LinkedList.cpp`, основную программу в файле `main.cpp`, файл `sdt.h` и ничего лишнего.

2. Поместить в скопированный каталог файл `make.bat`. Данный файл позволяет вызывать программу Make стандартной командой `make`, на самом деле обращаясь к `mingw32-make.exe` в составе MinGW-w64.
3. Поместить в скопированный каталог файл `lab06.bat` и запустить его. Данный файл настраивает окружение командной строки так, чтобы было удобно запускать средства автоматизации разработки ПО, вводя команды в открывающееся окно терминала.

Задание на лабораторную работу

1. Применить статическое тестирование (анализ исходного кода) для выявления и устранения заведомых ошибок в программе.
   1.1. Настроить расширение Code::Blocks для использования CppCheck из среды разработки. В диалоге *Settings / Environment…* на закладке *CppCheck* требуется указать полный путь к файлу `cppcheck.exe` (см. в каталоге `\Program Files`).
   1.2. Запустить статический анализатор пунктом меню *Plugins / CppCheck*.
   1.3. Просмотреть сообщения CppCheck, кроме относящихся к `sdt.h`, понять их смысл и при необходимости исправить, повторяя п. 1.2 для самопроверки.

   Указание. Статические анализаторы часто выдают предупреждения в случаях, когда ошибки нет (false-positive diagnostic), поэтому не во всех указанных CppCheck местах кода требуются исправления.

2. Настроить и применить генератор документации Doxygen.

   2.1. Создать файл `Doxyfile` с настройками Doxygen командой `doxygen -g -s`.

   Примечание. Без ключа `-s` команда `doxygen -g` добавляет в `Doxyfile` обильные комментарии, поясняющие каждый параметр, но затрудняющие редактирование. В качестве справки можно сгенерировать файл `doxygen.txt` с комментариями командой `doxygen -g doxygen.txt`.

   2.2. Сгенерировать документацию по имеющемуся коду командой `doxygen`.

   2.3. Просмотреть документацию в виде web-страниц, открыв файл `html/index.htm` (в «Проводнике»; здесь и далее пути даны относительно каталога проекта), после чего удалить сгенерированные файлы (каталоги `html/` и `rtf/` с их содержимым).

   2.4. Отредактировав файл `Doxyfile`, настроить генератор документации:

      1) помещать результаты генерации в каталог `docs` (`OUTPUT_DIRECTORY`);
      2) генерировать документацию только в виде web-страниц, отключив генерацию в формате LaTeX[1] и RTF (`GENERATE_LATEX`, `GENERATE_RTF`);
      3) использовать для документации заголовок «Учебный проект» и подзаголовок вида «Учебный проект Козлюка Дмитрия (А-02-09)», использовав свое имя и номер группы (`PROJECT_NAME`, `PROJECT_BRIEF`);
      4) записывать предупреждения Doxygen в текстовый файл (`WARN_LOGFILE`);
      5) генерировать документацию для внутренних нужд (`INTERNAL_DOCS`).

   Замечание. Если кодировка исходного кода отличается от UTF-8 (в Code::Blocks отображается слева внизу), потребуется изменить параметр `INPUT_ENCODING`.

   Указание. Параметры Doxygen задаются в `Doxyfile` и описаны в документации.

   Указание. При редактировании `Doxyfile` «Блокнотом» при `DOXYFILE_ENCODING` установленном в UTF-8 первый раз сохранять `Doxyfile` требуется через пункт меню «Сохранить как…» без расширения и в кодировке «Юникод (UTF-8)».

   2.5. Добавить подробные, осмысленные документирующие комментарии не менее, чем к одной структуре (или классу), трем функциям (или методам) и ко всем файлам.

   Указание. Потребуются команды `@file`, `@author`, `@date`, `@brief`, `@see`.

   2.6. Воспользоваться Doxygen из Code::Blocks при помощи расширения DoxyBlocks.

      2.6.1. С

      2.6.2. П

р

н

е

м

и

р

р

в

[1] LaTeX (читается «латéх») — издательская система для профессиональной подготовки книг, статей и других документов математического и технического содержания. Из текстовых файлов *.tex с разметкой генерируются, как правило, файлы PDF.

3. Выполнить анализ производительности методов класса `LinkedList`.
   3.1. Загрузить выполненное задание ЛР № 5 (исполняемый файл) `LinkedList.exe`, включающее многократные вызовы методов класса `LinkedList` и собранное для профилирования.
   3.2. Открыть собственный проект ЛР № 5 (или проект-заготовку при его отсутствии) и выбрать конфигурацию Release вместо Debug.
   3.3. В диалоге *Build options…*, вызываемом из контекстного меню проекта, включить режим *Profile code when executed [-pg]* для конфигурации сборки *Release*.
   3.4. Поместить файл `LinkedList.exe` в каталог `bin/Release` проекта.

   Внимание! Собирать (компилировать) проект в пункте 3 не нужно ни одного раза. Если сделать это, файл `LinkedList.exe` будет заменен результатом сборки, непригодным для профилирования, и его придется заменять снова.

   3.5. Запустить LinkedList.exe из «Проводника» (не из Code::Blocks), задать число замеров и дождаться окончания его работы. В том же каталоге (`bin/Release`) должен появиться файл `gmon.out`. Путем повторных запусков добиться времени профилирования около 100 с.
   3.6. В

   Указание. Если выполнить предыдущий пункт неправильно или не до конца, Code::Blocks отображает *важные* сообщения с предположениями о том, что могло быть сделано неверно.

   3.7. Вычислить время одного вызова каждого из десяти методов или функций, выполнение которых заняло наибольшее время. Сопоставить результаты с теоретическими знаниями о связанных списках и объяснить наблюдения.
4. Организовать автоматическую сборку программы без среды разработки.
   4.1. Организовать сборку программы системой Make, составив файл `Makefile`.
      4.1.1. Задать переменные с именами важных файлов, чтобы не повторять их внутри `Makefile` и иметь возможность легко изменить, например:

```
PROGRAM = Lab05.exe
SOURCES = LinkedList.cpp main.cpp
HEADERS = LinkedList.h sdt.h
```

      4.1.2. Задать переменную-список имен объектных файлов — каждый файл `*.cpp` преобразуется в объектный файл с тем же именем и расширением `*.o`:

```
OBJECTS = $(SOURCES:.cpp=.o)
```

      4.1.3. Основная цель `all` требует всех файлов исходного кода и файла программы:

```
all: $(SOURCES) $(HEADERS) $(PROGRAM)
```

      4.1.4. Файл программы получается путем компоновки всех объектных файлов в исполняемый, для чего нужны, очевидно, все объектные файлы:

```
$(PROGRAM): $(OBJECTS)
    g++ $(OBJECTS) -o $(PROGRAM)
```

4.1.5. Компиляция каждого файла исходного кода в одноименный объектный файл может выполняться следующим правилом:

```
.cpp.o:
    g++ -c -std=c++14 $< -o $@
```

4.1.6. Правило очистки (`clean`) ничего не требует и заключается в удалении выходных файлов — программы и объектных:

```
clean:
    del /Q $(PROGRAM)
    del /Q $(OBJECTS)
```

Внимание! У программы Make строгие требования к форматированию `Makefile`. Отступы обязательны и должны делаться табуляцией, а не пробелами. Пустые строки между целями обязательны и должны быть совершенно пустыми.

4.2. Убедиться в корректной настройке сборки, выполнив команду `make`, а затем проверить работу правила очистки командой `make clean`.

4.3. Автоматизировать запуск Make вспомогательных средств.

4.3.1. Настроить выполнение статического анализа кода при сборке.

1) добавить в `Makefile` цель `analysis` для получения `analysis.txt`; на основе файлов исходного кода и заголовочных файлов:

```
analysis: $(SOURCES) $(HEADERS) analysis.txt
```

2) для достижения цели `analysis.txt` запускать CppCheck командой

```
cppcheck $(SOURCES) LinkedList.h 2>analysis.txt
```

3) добавить цели `all` требование `analysis`:

```
all: $(SOURCES) $(HEADERS) $(PROGRAM) analysis
```

4) добавить цели `clean` новое действие — удаление `analysis.txt`.

Замечание. В шаге 2) файл `sdt.h` анализу намеренно не подвергается.

4.3.2. Выполнять генерацию документацию после сборки.

Указание. Действовать аналогично п. 4.2, добавив цель `docs` без требований и с командой `doxygen`, а также усовершенствовав правило очистки и основное правило `all`.

4.4. Обеспечить гибкость конфигурации сборки системой CMake.

Внимание! В дальнейшем `CMake` сгенерирует `Makefile` заново, поэтому написанный вручную `Makefile` следует скопировать в `Makefile.handmade`.

4.4.1. Составить файл `CMakeLists.txt` на основе примера, данного в лекции 10.

4.4.2. Добавить в `CMakeLists.txt` до команды `add_executable` строку

```
set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++14" )
```

Пояснение. CMake предоставляет специальные переменные для передачи параметров компилятору, так часто делают и в Makefile.

4.4.3. Сгенерировать `Makefile` на основе `CMakeLists.txt` командой

```
cmake -G "MinGW Makefiles"
```

4.4.4. Выполнить сборку программой `make` со сгенерированным `Makefile`.

4.5. Автоматизировать запуск вспомогательных средств средствами CMake.

Указание. Целесообразно воспользоваться второй формой `add_custom_command()`, например, для Doxygen:

```
add_custom_command(
    TARGET ${PROJECT_NAME}
    POST_BUILD
    COMMAND doxygen)
```

Вопросы для самопроверки

1. Что представляют собой и для чего нужны статические анализаторы кода?
2. Приведите два примера кода, в которых статический анализатор: а) обнаруживает реальную ошибку; б) совершает ложноположительное срабатывание (false-positive).
3. Каковы преимущества и недостатки генераторов документации к исходному коду по сравнению с написанием документации вручную?
4. Какие способы измерения производительности участков программы вам известны? Каковы преимущества и недостатки каждого из них?
5. Почему при анализе производительности желательно замерять время выполнения участка кода многократно? Какие искажения вносят во время выполнения вызовы функций?
6. Опишите этапы трансляции программы из исходного кода в исполняемый файл. Назовите конкретные программы, используемые в ЛР для каждого из этапов.
7. В чем состоит проблема повторного включения заголовочных файлов и каковы способы её решения? Почему они не нужны для файлов реализации (`*.cpp`)?
8. Почему определения шаблонов классов и функцией, используемых в разных единицах трансляции, требуется размещать в заголовочном файле?
9. Опишите назначение и работу программы `make` на примере программы из ЛР.
10. Опишите содержимое `Makefile`. Какова структура этого файла, какие языковые средства (переменные, ветвления, циклы и т. п.) доступны при его написании?
11. Каково назначение программы CMake и чем она полезна разработчику ПО? Приведите примеры проблем, существующих в `make` и решаемых CMake.