Компиляция и запуск программы на C

В этой лабораторной работе мы будем использовать программу командной строки g++ для компиляции программ на C. Самый простой способ запустить gcc следующий:

```
$ g++ program.c
```

Это компилирует program.c в исполняемый файл с именем a.out. Этот файл можно запустить с помощью следующей команды:

```
$ ./a.out
```

Исполняемый файл - a.out, почему добавляется ./? Когда вы хотите запустить исполняемый файл, вам нужно добавить путь к файлу, чтобы отличить вашу команду от такой команды, как python3, например. Точка указывает на текущую директорию. Таким образом, мы говорим, что запускаем a.out из текущей директории. Между прочим, двойные точки (..) исплользуются для именования родительской директории.

g++ имеет различные параметры командной строки, которые вам рекомендуется изучить. Особое внимание уделите параметрам `-Wall`, `-Werr`, `-O0`, `-O1`, `-O2`, `-O3` Однако в этой лабораторной работе мы будем использовать только `-o`, который используется для указания имени исполняемого файла, создаваемого gcc. Если вы не хотите, чтобы все ваши исполняемые файлы назывались a.out, вы можете использовать следующие команды, чтобы скомпилировать program.c в программу с именем program:

```
$ g++ -o program program.c
$ ./program
```

Необязательно: локальная установка компилятора C

Mac

Установите инструмент командной строки Xcode, запустив

```
$ xcode-select --install
```

Затем вы можете проверить, успешно ли установлен g++, запустив

```
$ g++ --version
```

Ubuntu / Debian

Установите пакет, необходимый для сборки, запустив

```
$ sudo apt install build-essential cgdb valgrind
```

При этом будет установлено множество пакетов, таких как g++ и make, а также инструменты отладки cgdb и valgrind. Если вам интересно, вот подробное описание пакета, необходимого для сборки, начиная с Ubuntu 18.04.

Windows

Для пользователей Windows рекомендуется использовать среду `cygwin`

Задание 1. Дебажим!

Отладчик, как следует из названия, - это программа, которая разработана специально для того, чтобы помочь отладить программу, найти логические ошибки и ошибки в вашем коде. У разных отладчиков разные функции, но обычно все отладчики могут выполнять следующие действия:

- Установить точку останова в своей программе. Точка останова - это определенная строка в вашем коде, где вы хотите остановить выполнение программы, чтобы вы могли посмотреть, что происходит поблизости.
- Построчно исполнить код программы. Код всегда выполняется последовательно, строка за строкой, но это происходит слишком быстро, чтобы мы могли понять, какие строки вызывают ошибки. Возможность поэтапного выполнения кода позволяет вам точно определить, что именно вызывает ошибку в вашей программе.

Для этого упражнения вам будет полезна справочная карта `GDB`, которая приложена отдельным файлом. `GDB` означает `GNU De-Bugger`. Скомпилируйте `hello.cpp` с флагом `-g`:

```
$ g++ -g -o hello hello.cpp
```

Это флаг указывает, что надо сохранить информацию в исполняемой программе, чтобы gdb мог ее понять. Теперь запустим наш отладчик `(c)gdb`:

```
$ gdb hello
```

Обратите внимание на то, что делает эта команда. Вы запускаете программу gdb передав путь к исполняемому файлу `hello` в качестве аргумента. Не пытайтесь запустить gdb с исходным кодом `hello.c`.

Примечание. gdb может не устанавливаться на компьютерах с macOS. Вы можете использовать lldb, еще один отличный отладчик. Команды немного отличаются, но есть руководства, которые помогут вам начать работу.

Задача

Пройдите через всю программу, выполнив следующие действия:

1. Установка точки останова на функции main
2. Исполните команду `run`
3. Исполните команду `step`

Введите `help` в gdb, чтобы узнать команды для выполнения этих действий, или воспользуйтесь справкой.

Примечание: cgdb против gdb

В этом упражнении мы используем gdb для отладки наших программ, но также стоит упомянуть cgdb. cgdb идентичен cgdb, за исключением того, что он предоставляет некоторые дополнительные приятные функции, которые делают его более удобным в использовании на практике.

В cgdb вы можете нажать `ESC`, чтобы перейти в окно кода (вверху), `i` чтобы вернуться в окно команд (внизу). В самом низу вы найдете окно команд, где вводятся команды `GDB`.

Пункт действия

Изучение этих команд окажется полезным для остальной части этой лабораторной работы и для вашей карьеры программиста на языке C++ в целом. В текстовом файле с именем gdb.txt ответьте на следующие вопросы.

1. Когда вы находитесь в сеансе gdb, как вы устанавливаете аргументы, которые будут передаваться программе при ее запуске?
2. Как создать точку останова?
3. Как выполнить следующую строку кода C в программе после остановки в точке останова?

4. Если следующая строка кода является вызовом функции, вы выполните весь вызов функции сразу, если воспользуетесь своим ответом на # 3. (Если нет, рассмотрите другую команду для # 3!) Как вы сообщите GDB, что хотите вместо этого отладить код внутри функции (т.е. перейти внутрь функции)?
5. Как продолжить выполнение программы после остановки на точке останова?
6. Как вы можете распечатать значение переменной (или даже выражение типа 1 + 2) в gdb?
7. Как настроить gdb так, чтобы он отображал значение переменной после каждого шага?
8. Как вы показываете список всех переменных и их значений в текущей функции?
9. Как выйти из GDB?

Задание 2. Интерактивная отладка

Давайте посмотрим, что произойдет, если ваша программа требует ввода данных пользователем, и вы попытаетесь запустить её в `GDB`. Во-первых, соберите и запустите программу, определенную в `interactive_hello.cpp`, чтобы поговорить с слишком дружелюбной программой.

```
$ g++ -g -o int_hello interactive_hello.cpp
$ ./int_hello
```

Теперь попробуем отладить его (хотя на самом деле ошибок нет).

```
$ gdb int_hello
```

Что происходит, когда вы пытаетесь запустить программу до конца исполнения? Цель этого задания - избавить вас от боязни запуска отладчика, даже если вашей программе требуется ввод данных пользователем. Оказывается, вы можете отправлять текст на стандартный ввод `stdin`, файловый поток, читаемый через объект `std::cin`, в этой глупой программе, с некоторыми специальными символами прямо из командной строки.

Подсказка 1. Если вы создаете текстовый файл, содержащий введенные вами данные, вы на правильном пути! *Подсказка 2.* Помните, что вы также можете запускать программы с аргументами командной строки (включая символы перенаправления) из `GDB`!

Задание 3. Valgrind/Sanitizer

Даже с отладчиком мы не сможем отловить все ошибки. Некоторые ошибки мы называем Борбагами, что означает, что они надежно проявляются при четко определенном, но, возможно, неизвестном наборе условий. Другие ошибки мы называем Гейзенбагами, и вместо того, чтобы быть четко определенными, они исчезают или изменяют свое поведение, когда их пытаются изучить. Мы можем обнаружить первый тип с помощью отладчиков, но второй тип может ускользнуть от нас, потому что они (по крайней мере, в `C/C++`) часто возникают из-за неправильного управления памятью.

Помните, что в отличие от других языков программирования, `C` требует, чтобы вы (программист) вручную управляли своей памятью.

Чтобы помочь отловить этих Гейзенбагов, мы можем использовать под названием `Valgrind` или встроенные в `g++` санитайзеры. Мы рассмотрим оба метода, вы можете выбрать наиболее удобный для выполнения задания. `Valgrind` - это программа, которая имитирует ваш процессор и отслеживает обращения к вашей памяти. Это замедляет выполняемый вами процесс (поэтому мы, например, не всегда запускаем все исполняемые файлы внутри `Valgrind`), но также может выявить ошибки, которые могут отображать только видимое неправильное поведение при уникальном стечении обстоятельств.

Примечание. `Valgrind` доступен для установки в большинстве Unix-подобных дистрибутивов, а также в macOS, однако возможны проблемы совместимости между `Valgrind` и последними версиями macOS. Если у вас не получается запустисть `Valgrind` из `homebrew`, придется собрать вручную на своей машине следую инструкциям.

В этом упражнении мы продемонстрируем два разных примера выходных данных `Valgrind` и рассмотрим, как каждый из них может быть полезен.

Сначала соберите два новых исполняемых файла, `segfault_ex` из `segfault_ex.cpp` и `no_segfault_ex` из `no_segfault_ex.cpp` (используйте флаг `-o`). На этом этапе вы сможете использовать `g++` для успешной сборки этих исполняемых файлов.

Теперь попробуем запустить исполняемые файлы. Какие результаты вы наблюдаете?

Начнем с `segfault_ex`. Вы должны были наблюдать ошибку сегментации (`segfault`), которая возникает, когда программа вылетает из-за попытки получить доступ к памяти, которая ей недоступна (подробнее об этом вы узнаете позже в курсе; на самом деле это артефакт из раннего проектирования памяти. Сегодня мы работаем с «выгружаемой памятью» вместо «сегментированной памяти», но сообщение об ошибке осталось).

Этот файл исходный код очень мал, поэтому вы должны иметь возможность открыть файл и понять, что вызывает `segfault`. Сделайте это сейчас, но не

меняйте файл. Почему возникает segfault?

Теперь давайте разберемся, что делать, если у нас очень большой файл и нужно найти segfault. Вот где нам поможет Valgrind. Для запуска программы в Valgrind используйте команду:

```
$ valgrind ./segfault_ex
```

Это должно привести к тому, что Valgrind выведет информацию о том, где произошел незаконный доступ. Сравните эти результаты с вашими предположениями. Как Valgrind может помочь вам устранить ошибку сегментации в будущем?

Теперь попробуйте запустить Valgrind на no_segfault_ex. Программа не должна была дать сбой, но проблема с файлом все еще существует. Valgrind может помочь нам найти (на первый взгляд невидимую) проблему.

К сожалению, здесь вы увидите, что Valgrind, похоже, не может сказать вам, где именно возникает проблема. Используйте сообщение, предоставленное Valgrind, чтобы определить, какая переменная вызывает некорректное поведение, а затем попытайтесь сделать вывод, что должно было произойти (Подсказка: что такое неинициализированное значение?).

Другим подходом к исследованию таких ошибок является использование санитайзеров. Для этого может понадобиться установить специальные библиотеки:

```
sudo dnf install libasan-static # centos/fedora
sudo apt-get install -y libasan2 # Ubuntu
```

После этого, можно собирать программу со специальным ключом g++, например:

```
g++ segfault_ex.cpp -fsanitize=address -static-libasan -g -o segfault_ex
```

Флаг -fsanitize=address включает санитайзер, -static-libasan статически связывает вашу программу с библиотекой санитайзера. Флаг -g нужен для отображения символов отладки при запуске.

После того как программа собрана с включенным санитайзером, она запускается обычным способом:

```
./sefgault_ex
```

Надеюсь, пройдя этот пример, вы сможете понять и ответить на следующие вопросы:

1. Почему важны иснтрументы вроде `Valgrind` и санитайзеров, какая в них польза?
2. Как запустить программу в `Valgrind`?
3. Как использовать санитайзер `G++`?
4. Как вы интерпретируете сообщения об ошибках?
5. Почему неинициализированные переменные могут приводить к появлению Гейзенбагов?

Это черезвычайно важные инструменты, которые очень полезны при написании кода на `C\C++`. Будьте готовы ответить на следующие вопросы:

1. Почему не произошел segfault программы no_segfault_ex?
2. Почему no_segfault_ex выдает несогласованные выходные данные?
3. Почему указан неправильный размер? Как можно было исправить код используя sizeof?

Задача

Исправьте ошибки с помощью `Valgrind` или санитайзера.

Задание 4. Makefile. Собираем проще.

Как вы уже могли заметить, компиляция программ C в терминале - утомительная и трудоемкая операция, требующая выполнения нескольких команд с большим количеством аргументов. Хотя это возможно для простых программ, для более крупных и сложных программ с десятками файлов и зависимостей это быстро становится довольно неудобным.

Для сборки больших и сложных обычно используется так называемый «make-файл», который облегчает компиляцию. `Makefile` - это текстовый файл (буквально названный `Makefile`) в каталоге кода, который содержит набор правил, каждое из которых имеет команды, компилирующие для них C/C++ программу. Каждый make-файл может содержать несколько правил, каждое из которых компилирует одну или несколько целей, исполняемый файл или библиотеку, или выполняет другую задачу. Чтобы скомпилировать цель, программисту просто нужно ввести `make` в командный терминал.

Задача

Взгляните на `Makefile`, включенный в эту лабораторную работу, и попытайтесь ответить на следующие вопросы в `make.txt`. Обязательно используйте Интернет, чтобы выяснить некоторые из этих вопросов, но ответы пишите своими словами, как вы сами понимаете.

1. Какая цель является частью правила, удаляющего все скомпилированные программы?
2. Какая цель является частью правила, которое создает все скомпилированные программы?
3. Какой компилятор сейчас используется?
4. Какой стандарт C мы используем в настоящее время?
5. Как мы можем ссылаться на переменную «FOO» в make-файле?
6. Для чего используется переменная `CFLAGS`?
7. Какая строка создает программу `bit-ops` из ее объектных файлов? (Укажите номер строки.)

На самом деле, для серьезных проектов даже make-файлы оказываются слишком сложными и трудночитаемыми. Поэтому разработаны другие системы сборки проектов, такие как `cmake`.

Задание 5: Битовые операции

В этом упражнении вы завершите `bit_ops.cpp`, реализовав функции манипулирования битами `get_bit`, `set_bit` и `flip_bit`, которые представлены ниже. В комментариях файла `bit_ops.cpp` описано ожидаемое поведение этих функций. Вы можете использовать ТОЛЬКО побитовые операции, такие как И `&`, ИЛИ `|`, XOR `^`, НЕ `~`, сдвиг влево `<<` и сдвиг вправо `>>`. Вы не можете использовать циклы `for` / `while` или условные операторы. Вы также не можете использовать деление по модулю, деление, сложение, вычитание или умножение в этом задании.

```
unsigned get_bit(unsigned x, unsigned n);
void set_bit(unsigned * x, unsigned n, unsigned v);
void flip_bit(unsigned * x, unsigned n);
```

Задача: Завершите реализацию `get_bit`, `set_bit` и `flip_bit`.

Выполнив эти функции, вы можете скомпилировать и запустить свой код с помощью следующих команд:

```
$ make bit_ops
$ ./bit_ops
```

Это распечатает результат выполнения тестов. Очевидно, все тесты должны проходить успешно.