26.05.2025

Интеграция с другими языками. Модульное тестирование и отладка. Системы сборки.

Филиппов Михаил Витальевич

m.filippov@g.nsu.ru

89232283872

Императивное программирование, 2024-2025

Новосибирский государственный университет

*НАСТОЯЩАЯ НАУКА

Давайте познакомимся

Филиппов Михаил Витальевич

- Окончил магистратуру ФФ НГУ
- Окончил аспирантуру ИТ СО РАН
- Являюсь м.н.с. ИТ СО РАН
- 7+ лет опыт в программировании C/C++

План лекции

Интеграция с другими языками

35 минут

Модульное тестирование и отладка

35 минут

Системы сборки

20 минут

План лекции

Интеграция с другими языками

35 минут

Модульное тестирование и отладка

35 минут

Системы сборки

20 минут

Введение

Навыки написания программ и библиотек на языке C могут оказаться еще полезнее, чем можно было бы ожидать. Благодаря важной роли в разработке операционных систем язык C проникает в другие области. Написанные на нем библиотеки потенциально могут загружаться и применяться в других языках программирования. Пользуясь преимуществами высокоуровневых языков, вы можете задействовать в их средах эффективный и быстрый код, написанный на C.

Что делает интеграцию возможной

Язык C кардинально изменил разработку операционных систем. Но, помимо этого, он также сделал возможным создание поверх него других языков программирования общего назначения. В наши дни мы называем их высокоуровневыми. В большинстве своем компиляторы этих языков написаны на C, а остальные разработаны с помощью других инструментов и компиляторов, которые тоже написаны на C.

Язык программирования общего назначения, неспособный применять или предоставлять возможности операционной системы, совершенно бесполезен. Вы можете писать на нем, однако написанное нельзя будет выполнить ни в одной ОС. Такой язык можно задействовать в теоретических целях, но в промышленном смысле он ни на что не годится. Поэтому языки, и особенно их компиляторы, должны уметь генерировать рабочие программы.

И здесь на помощь приходит C. Возможности Unix-подобных операционных систем доступны в виде API, который предоставляет стандартная библиотека C. Если компилятор хочет создать рабочую программу, то должен дать возможность скомпилированной программе работать со стандартной библиотекой C опосредованным образом. Независимо от языка и наличия в нем скомпилированной стандартной библиотеки, когда написанная на нем программа обращается к той или иной функции (такой как открытие файла), этот запрос должен быть передан стандартной библиотеке C, откуда он может быть направлен на выполнение в ядро. Примером этого служит язык Java, предоставляющий пакет Java Standard Edition (Java SE).

В заключение отмечу: в качестве библиотек, которые загружаются в другие языки, могут выступать только разделяемые объектные файлы.

Получение необходимых материалов

Пример: lection_33_integration-with-other-languages

В целях сборки исходного кода на вашем компьютере должны быть установлены Java Development Kit (JDK), Python и Golang. В зависимости от вашей операционной системы (Linux или macOS) и дистрибутива Linux, команды установки могут отличаться.

Библиотека для работы со стеком

```
typedef int bool_t;
typedef struct {
  char *data;
  size_t len;
} value_t;
typedef struct cstack_type cstack_t;
typedef void (*deleter_t)(value_t *value);
value_t make_value(char *data, size_t len);
value_t copy_value(char *data, size_t len);
void free_value(value_t *value);
cstack_t *cstack_new();
void cstack_delete(cstack_t *);
// Функции поведения
void cstack_ctor(cstack_t *, size_t);
void cstack_dtor(cstack_t *, deleter_t);
size_t cstack_size(const cstack_t *);
bool_t cstack_push(cstack_t *, value_t value);
bool_t cstack_pop(cstack_t *, value_t *value);
void cstack_clear(cstack_t *, deleter_t);
```

Небольшая библиотека, которая будет загружаться и использоваться программами, написанными на других языках помимо C. В основе библиотеки лежит класс Stack, который предоставляет базовые операции для работы с объектами в стеке, такие как push и pop. Данные объекты создаются и уничтожаются самой библиотекой; для этого предусмотрены конструктор и деструктор.

Пример: lection_33_integration-with-other-languages, файл cstack.h

Библиотека для работы со стеком

Пример: lection_33_integration-with-other-languages, файл build_linux.sh

```
set -x
gcc -c -g -fPIC cstack.c -o cstack.o
gcc -shared cstack.o -o libcstack.so
gcc -c -g cstack_tests.c -o tests.o
gcc tests.o -lcstack -L. -o cstack_tests.out
```

Пример: lection_33_integration-with-other-languages, файл run_tests_linux.sh

```
set -x
export CSTACK_LIB_PATH=.
LD_LIBRARY_PATH=$CSTACK_LIB_PATH
./cstack_tests.out
```

Итак, мы получили объектный файл разделяемой библиотеки. Теперь можно написать программы на других языках, которые будут его загружать.

Сборка библиотеки для работы со стеком и создание разделяемого объектного файла в Linux + запуск тестов

```
$ ./build_linux.sh
+ gcc -c -g -fPIC cstack.c -o cstack.o
+ gcc -shared cstack.o -o libcstack.so
+ gcc -c -g cstack_tests.c -o tests.o
+ gcc tests.o -lcstack -L. -o cstack_tests.out
```

```
$ ./run_tests_linux.sh
++ export CSTACK_LIB_PATH=.
++ CSTACK_LIB_PATH=.
++ LD_LIBRARY_PATH=.
++ ./cstack_tests.out
All tests were OK.
```

Библиотека для работы со стеком

Выполнение тестов с использованием утилиты valgrind

```
$ LD_LIBRARY_PATH=$PWD valgrind --leak-check=full ./cstack_tests.out
==1841== Memcheck, a memory error detector
==1841== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1841== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==1841== Command: ./cstack_tests.out
==1841==
All tests were OK.
==1841==
==1841== HEAP SUMMARY:
==1841==     in use at exit: 0 bytes in 0 blocks
==1841==   total heap usage: 10 allocs, 10 frees, 2,676 bytes allocated
==1841==
==1841== All heap blocks were freed -- no leaks are possible
==1841==
==1841== For lists of detected and suppressed errors, rerun with: -s
==1841== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

У нас нет никаких утечек памяти. Это позволяет нам чувствовать определенную уверенность по отношению к написанной нами библиотеке. Таким образом, поиск первопричин любых проблем с памятью следует начинать со среды выполнения, в которой загружена библиотека.

Интеграция с C++

Интеграцию с C++ можно считать самой простой. C++ — своего рода объектно-ориентированное расширение C. Компиляторы этих языков генерируют похожие объектные файлы. Поэтому программе на C++ проще загружать разделяемые библиотеки C по сравнению с другими языками. Иными словами, программа на C++ способна в равной степени загружать объектные файлы, написанные на C и C++. Единственным проблемным моментом в некоторых случаях может быть декорирование имен в C++,

Декорирование имен в C++

В языке C++ имена символов, относящихся к функциям (как к глобальным, так и к методам классов), декорируются. В основном так поддерживаются возможности наподобие пространства имен и перегрузка функций, которых нет в C. При сборке кода на языке C с использованием компилятора C++ данная функция включена по умолчанию, поэтому мы ожидаем увидеть декорированные имена символов.

```
int add(int a, int b)
{
    return a + b;
}
```

```
$ clang -c test_1.c -o test_1.o
$ nm test_1.o
0000000000000000 T add
$ clang++ -c test_1.c -o test_1.o
clang++: warning: treating 'c' input as 'c++' when in C++ mode, this
behavior is deprecated [-Wdeprecated]
$ nm test_1.o
0000000000000000 T _Z3addii
```

Компилятор clang++ сгенерировал предупреждение о том, что в ближайшем будущем поддержка компиляции кода на C так, будто он написан на C++, исчезнет. Имя символа, сгенерированного для представленной выше функции, было декорировано и отличается от того, которое было получено при использовании clang.

Декорирование имен в C++

Чтобы устранить эту проблему, код на языке C необходимо завернуть в специальный блок, который не даст компилятору C++ декорировать имена символов. В результате при компиляции с помощью clang и clang++ названия символов будут совпадать. Эта функция помещается в блок extern "C" { ... }, только если у нас уже определен макрос __cplusplus. Наличие последнего — признак того, что код собирается компилятором C++.

```
#ifdef __cplusplus
extern "C"
{
#endif
    int add(int a,
int b)
    {
        return a + b;
    }
#ifdef __cplusplus
}
#endif
```

```
$ clang -c test_1.c -o test_1.o
$ nm test_1.o
0000000000000000 T add
$ clang++ -c test_1.c -o test_1.o
clang++: warning: treating 'c' input as 'c++' when in C++ mode, this
behavior is deprecated [-Wdeprecated]
$ nm test_1.o
0000000000000000 T add
```

Что касается нашей библиотеки для работы со стеком, все объявления необходимо поместить в блок extern "C" { … }. При компоновке программы на C++ с нашей библиотекой символы можно будет найти внутри libcstack.so (или libcstack.dylib). extern "C" — это спецификация компоновки. Подробности по ссылкам: 1, 2.

Код на C++

Класс языка C++, в который завернута функциональность, предоставляемая библиотекой для работы со стеком (c++/Stack.cpp)

```
#include "cstack.h"
...
template<typename T>
class Stack {
...
  size_t Size() {
    return cstack_size(mStack);
  }
  void Push(const T& pItem)
...
private:
  cstack_t* mStack;
};
template<>
value_t CreateValue(const std::string& pValue)
  ...
int main(int argc, char** argv) {
  Stack<std::string> stringStack(100);
  stringStack.Push("Hello");
...
  return 0;
}
```

Теперь можем написать программу на C++, которая будет использовать библиотеку для работы со стеком. Для начала обернем нашу библиотеку в класс, который является главным составным компонентом объектноориентированной программы на C++. Доступ к возможностям библиотеки лучше предоставлять в объектно-ориентированной манере, вместо того чтобы вызывать функции C напрямую.

Код на C++

Пример: lection_33_integration-with-other-languages, файл c++/build_linux.sh

```
set -x
g++ -c -g -std=c++11 -I$PWD/.. Stack.cpp -o Stack.o
g++ -L$PWD/.. Stack.o -lcstack -o cstack_cpp.out
```

Пример: lection_33_integration-with-other-languages, файл c++/ run_tests_linux.sh

```
set -x
LD_LIBRARY_PATH=$PWD/.. ./cstack_cpp.out
```

Сборка библиотеки для работы со стеком и создание разделяемого объектного файла в Linux + запуск тестов

```
$ cd c++
$ ./build_linux.sh
++ g++ -c -g -std=c++11 -I/mnt/
c/Users/.../c++/.. Stack.cpp -o Stack.o
++ g++ -L/mnt/c/ c/Users/.../c++/..
Stack.o -lcstack -o cstack_cpp.out
```

```
$ ./run_linux.sh
$ ++ LD_LIBRARY_PATH=/mnt/c/Users/.../c++/..
++ ./cstack_cpp.out
Stack size: 3
Popped > !
Popped > World
Popped > Hello
Stack size after pops: 0
Stack size before clear: 2
Stack size after clear: 0
```

Код на C++

Сборка и запуск кода на C++ с использованием утилиты valgrind

```
$ LD_LIBRARY_PATH=$PWD/.. valgrind --leak-check=full ./cstack_cpp.out
==2116== Memcheck, a memory error detector
==2116== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2116== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==2116== Command: ./cstack_cpp.out
==2116==
Stack size: 3
Popped > !
Popped > World
Popped > Hello
Stack size after pops: 0
Stack size before clear: 2
Stack size after clear: 0
==2116==
==2116== HEAP SUMMARY:
==2116==     in use at exit: 0 bytes in 0 blocks
==2116==   total heap usage: 9 allocs, 9 frees, 76,398 bytes allocated
==2116==
==2116== All heap blocks were freed -- no leaks are possible
==2116==
==2116== For lists of detected and suppressed errors, rerun with: -s
==2116== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Интеграция с Java

Java-программы собираются компилятором Java в специальный байт-код, аналогичный формату объектных файлов ABI (Application Binary Interface — двоичный интерфейс приложений). Файлы с байт-кодом Java нельзя выполнять как обычные программы, для работы им нужна специальная среда.

Байт-код Java может выполняться только в виртуальной машине Java (Java Virtual Machine, JVM). Она представляет собой процесс, который имитирует рабочую среду для байт-кода Java. Обычно написана на C или C++ и способна загружать и использовать стандартную библиотеку C вместе со всеми функциями, которые та предоставляет.

В байт-код Java можно компилировать не только сам язык Java, но и, к примеру, Scala, Kotlin и Groovy. Таким образом, все эти языки программирования могут работать внутри JVM. Их обычно называют языками JVM.

Написание кода на Java

Представьте, что у нас есть проект на C, собранный в разделяемую объектную библиотеку. Мы хотим подключить его к Java и воспользоваться его функциями. К счастью, для компиляции кода на языке Java вовсе не требуется наличие кода на C (исходного или скомпилированного). В Java эти две части проекта полностью отделены друг от друга с помощью машинно-зависимых методов (native methods). Конечно, программу не получится запустить с одним лишь кодом на Java, поскольку мы вызываем функции C, что требует предварительной загрузки файла с разделяемой объектной библиотекой. Мы предоставим все инструкции и исходники, необходимые для выполнения Java-программы, которая успешно загружает разделяемую библиотеку и вызывает ее функции.

Для загрузки разделяемых объектных файлов в Java используется JNI (Java Native Interface). Отмечу, что JNI не входит в состав данного языка программирования; скорее, это часть спецификации JVM, вследствие чего импортированную библиотеку можно будет задействовать во всех языках JVM, таких как Scala.

JNI использует машинно-зависимые методы, не имеющие никаких определений на языке Java; их реальные определения написаны на C или C++ и находятся во внешних разделяемых библиотеках. Можно сказать, что машинно-зависимые методы — это своеобразные порты, через которые программы, написанные на Java, могут общаться с внешним миром и выходить за пределы JVM.

Класс NativeStack
(java/src/com/packt/example_c/c_java/ex/Main.java)

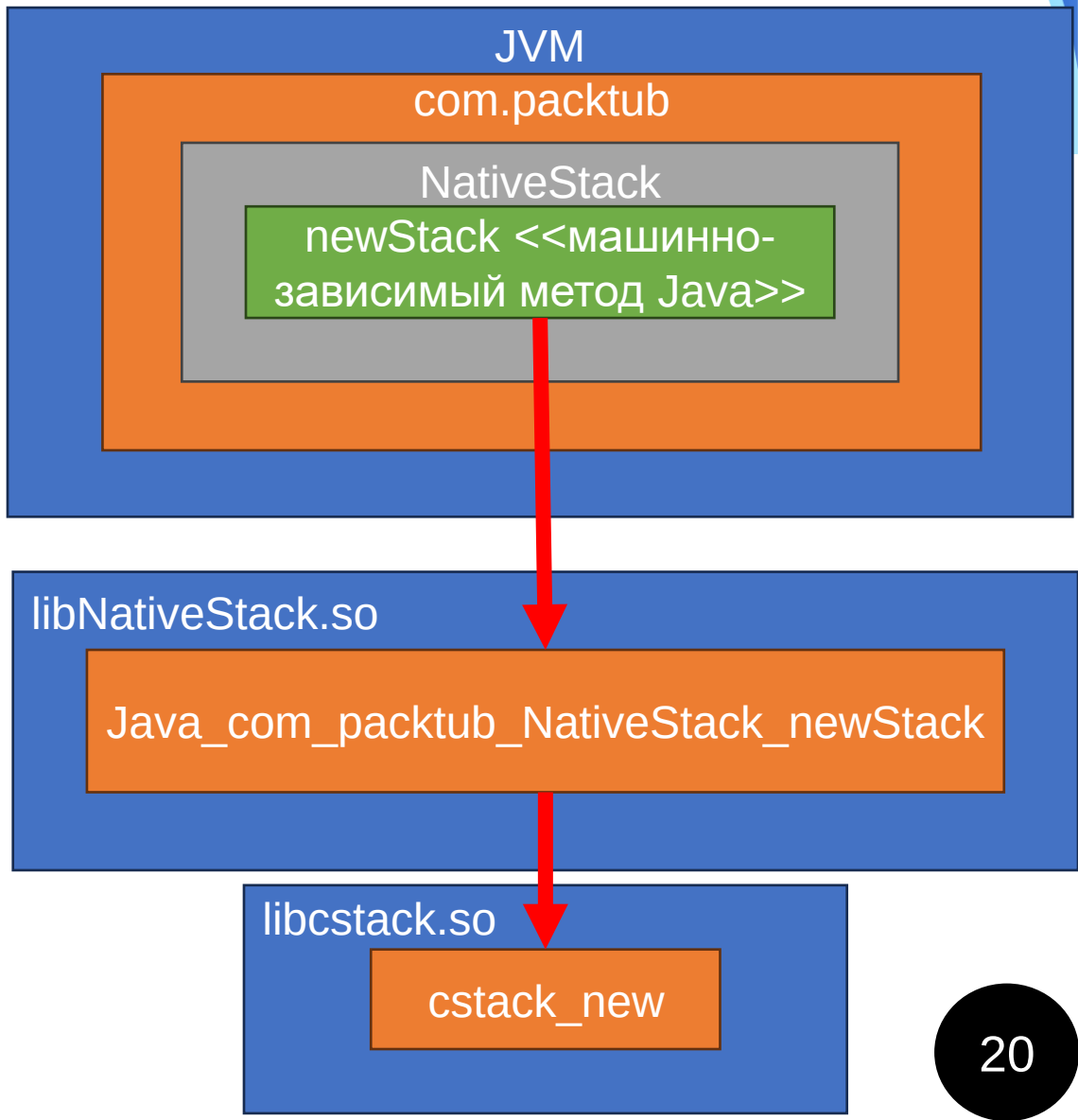
Написание кода на Java

```
class NativeStack {
  static {
    System.loadLibrary("NativeStack");
  }
  public static native long newStack();
  public static native void deleteStack(long stackHandler);
  public static native void ctor(long stackHandler, int maxSize);
  public static native void dtor(long stackHandler);
  public static native int size(long stackHandler);
  public static native void push(long stackHandler, byte[] item);
  public static native byte[] pop(long stackHandler);
  public static native void clear(long stackHandler);
}
interface Marshaller<T> {
  byte[] marshal(T obj);
  T unmarshal(byte[] data);
}
class Stack<T> implements AutoCloseable {
private Marshaller<T> marshaller;
  private long stackHandler;
  public Stack(Marshaller<T> marshaller) {
    this.marshaller = marshaller;
    this.stackHandler = NativeStack.newStack();
    NativeStack.ctor(stackHandler, 100);
  }
...
```

Методы соответствуют функциям нашей библиотеки на языке C. ***Обратите внимание:*** роль первого операнда играет переменная long. Она содержит адрес памяти, прочитанный из машинно-зависимой библиотеки, и выступает указателем, который нужно передавать другим методам в целях обозначения экземпляра стека. Для написания приведенного выше класса нам не нужен уже готовый и полностью рабочий разделяемый объектный файл. Все, что нам требуется, — это список объявлений, необходимых для определения API стека.

Написание машинно-зависимой части

Самым важным аспектом предыдущего раздела была концепция машинно-зависимых методов. Они объявляются в Java, но их определения находятся в разделяемой объектной библиотеке за пределами JVM. Но как виртуальная машина находит эти определения? Ответ прост: по именам символов в загруженных объектных файлах. JVM извлекает имя символа для каждого машинно-зависимого метода с учетом различных свойств, таких как пакет, содержащий и его имя. Затем в загруженных разделяемых библиотеках ищется соответствующий символ, и в случае неудачи возвращается ошибка.JVM вынуждает нас использовать для функций, которые входят в состав загружаемого объектного файла, символы с определенными именами. Однако при создании нашей библиотеки для работы со стеком мы не следовали никаким соглашениям об именовании. Таким образом, JVM не сможет найти функции, предоставляемые нашей библиотекой. Поэтому мы должны найти другой путь. Библиотеки C обычно пишут без расчета на то, что они будут применяться в среде JVM.



Написание машинно-зависимой части

Представьте, что мы хотим вызвать функцию C, func, из Java, и определение данной функции находится в разделяемом объектном файле libfunc.so. В коде на языке Java также имеется класс Clazz с методом doFunc. Мы знаем, что при поиске определения машинно-зависимой функции doFunc JVM будет искать символ Java_Clazz_doFunc. Поэтому создадим промежуточную разделяемую библиотеку libNativeLibrary.so, содержащую именно такой символ, который ищет JVM. Затем внутри данной функции мы сделаем вызов func. Можно сказать, что Java_Clazz_doFunc играет роль реле, делегируя вызов библиотеке C и в конечном счете функции func.

Чтобы у нас совпадали имена символов Java, можно воспользоваться заголовочным файлом C, который компилятор Java обычно генерирует из машинно-зависимых методов, найденных в коде на Java. Таким образом, нам остается только написать определения для функций, содержащихся в данном заголовочном файле. Это позволяет не ошибиться в именах символов, которые в конечном счете будет искать JVM.

Компиляция файла Main.java и генерация заголовка из найденных в нем машинно-зависимых методов

Компиляция файла Main.java и генерация заголовка из найденных в нем машинно-зависимых методов

build.sh. Как видите, мы передали компилятору Java, javac, параметр -h.

```
set -x
mkdir -p build/headers
mkdir -p build/classes
javac -cp src -h build/headers -d build/classes
src/com/packt/example_c/c_java/ex/Main.java
```

Написание машинно-зависимой части

```
$ cd java
$ ./build.sh
++ mkdir -p build/headers
++ mkdir -p build/classes
++ javac -cp src -h build/headers -d build/classes
src/com/packt/example_c/c_java/ex/Main.java
$ tree build
build
├── classes
│   └── com
│       └── packt
│           └── example_c
│               └── c_java
│                   └── ex
│                       ├── Main.class
│                       ├── Marshaller.class
│                       ├── NativeStack.class
│                       ├── Stack.class
│                       └── StringMarshaller.class
└── headers
    └── com_packt_example_c_c_java_ex_NativeStack.h
8 directories, 6 files
```

Утилита tree отображает содержимое каталога build в виде дерева. Обратите внимание на файлы .class. Они содержат байт-код Java, который будет использоваться при загрузке этих классов в экземпляр JVM.

Помимо файлов с классами, мы видим заголовочный файл, com_packt_extreme_c_ NativeStack.h, который содержит соответствующие определения функций языка C для машинно-зависимых методов из класса NativeStack.

Написание машинно-зависимой части

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class com_packt_example_c_c_java_ex_NativeStack */

#ifndef _Included_com_packt_example_c_c_java_ex_NativeStack
#define _Included_com_packt_example_c_c_java_ex_NativeStack
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     com_packt_example_c_c_java_ex_NativeStack
 * Method:    newStack
 * Signature: ()J
 */
JNIEXPORT jlong JNICALL
Java_com_packt_example_1c_c_1java_ex_NativeStack_newStack
  (JNIEnv *, jclass);
...
```

(Неполное) содержимое заголовочного файла, сгенерированного в соответствии с JNI

Написание машинно-зависимой части

```
#include <jni.h>
#ifndef _Included_com_packt_NativeStack
#define _Included_com_packt_NativeStack
#define JNI_FUNC(n) Java_com_packt_example_1c_c_1java_ex_NativeStack_##n
#ifdef __cplusplus
extern "C"
{
#endif
    JNIEXPORT jlong JNICALL JNI_FUNC(newStack)(JNIEnv *, jclass);
    JNIEXPORT void JNICALL JNI_FUNC(deleteStack)(JNIEnv *, jclass, jlong);
    JNIEXPORT void JNICALL JNI_FUNC(ctor)(JNIEnv *, jclass, jlong, jint);
    JNIEXPORT void JNICALL JNI_FUNC(dtor)(JNIEnv *, jclass, jlong);
    JNIEXPORT jint JNICALL JNI_FUNC(size)(JNIEnv *, jclass, jlong);
    JNIEXPORT void JNICALL JNI_FUNC(push)(JNIEnv *, jclass, jlong, jbyteArray);
    JNIEXPORT jbyteArray JNICALL JNI_FUNC(pop)(JNIEnv *, jclass, jlong);
    JNIEXPORT void JNICALL JNI_FUNC(clear)(JNIEnv *, jclass, jlong);
#ifdef __cplusplus
}
#endif
#endif
```

Модифицированная версия сгенерированного заголовочного файла JNI (java/native/NativeStack.h)

Написание машинно-зависимой части

Сборка промежуточной разделяемой библиотеки libNativeStack.so

```
set -x
g++ -c -g -fPIC -I$PWD/../.. -I$JAVA_HOME/include \
  -I$JAVA_HOME/include/linux NativeStack.cpp -o NativeStack.o
g++ -shared -L$PWD/../.. NativeStack.o -lcstack -o
libNativeStack.so
```

Итоговый разделяемый объектный файл компонуется с библиотекой для работы со стеком, libcstack.so. Это значит, что чтобы библиотека libNativeStack.so могла работать, нужно загрузить libcstack.so. Следовательно, JVM загружает по очереди эти два файла, и в итоге код на Java сможет взаимодействовать с машинно-зависимой частью проекта, что позволит нам выполнить нашу Java-программу.

Сборка промежуточной разделяемой библиотеки libNativeStack.so

```
$ cd java/native
$ export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
$ ./build_linux.sh
++ g++ -c -g -fPIC -I/mnt/c/Users/Mfili/.../lection_33_integration-with-other-
languages/java/native/../.. -I/usr/lib/jvm/java-11-openjdk-amd64/include -
I/usr/lib/jvm/java-11-openjdk-amd64/include/linux NativeStack.cpp -o NativeStack.o
++ g++ -shared -L/mnt/c/Users/Mfili/.../lection_33_integration-with-other-
languages/java/native/../.. NativeStack.o -lcstack -o libNativeStack.so
```

Написание машинно-зависимой части

Выполнение демонстрационного примера на Java

```
set -x
LD_LIBRARY_PATH=$PWD/.. java -Djava.library.path=$PWD/native -cp
build/classes com.packt.extreme_c.ch21.ex1.Main
```

Выполнение демонстрационного примера на Java

```
$ cd java
$ ./run_linux.sh
++ LD_LIBRARY_PATH=/mnt/c/Users/Mfili/.../lection_33_integration-with-other-
languages/java/..
++ java -Djava.library.path=/mnt/c/Users/Mfili/.../lection_33_integration-with-
other-languages/java/native -cp build/classes com.packt.example_c.c_java.ex.Main
Size after pushes: 3
!
World
Hello
Size after pops: 0
Size after before clear: 2
Size after clear: 0
```

Интеграция с Python

Python — интерпретируемый язык программирования. Это значит, написанный на нем код анализируется и выполняется промежуточной программой, известной как интерпретатор. Если мы будем использовать внешнюю скомпилированную разделяемую библиотеку, то интерпретатор ее загрузит и сделает доступной в исходном коде. У Python есть специальный фреймворк для загрузки внешних разделяемых библиотек под названием ctypes.

Интеграция с Python

Код, который относится к ctypes и делает библиотеку для работы со стеком доступной остальному коду на Python **(python/stack.py)**

```
import platform
from ctypes import *

class value_t(Structure):
  _fields_ = [("data", c_char_p), ("len", c_int)]

class _NativeStack:
  def __init__(self):
    self.stackLib = cdll.LoadLibrary(
            "libcstack.dylib" if platform.system() == 'Darwin'
            else "libcstack.so")

    # value_t make_value(char*, size_t)
    self._makevalue_ = self.stackLib.make_value
    self._makevalue_.argtypes = [c_char_p, c_int]
    self._makevalue_.restype = value_t
 ...
```

Интеграция с Python

Класс Stack на Python, который использует функции C, загруженные из библиотеки для работы со стеком **(python/stack.py)**

```
class Stack:
  def __enter__(self):
    self._nativeApi_ = _NativeStack()
    self._handler_ = self._nativeApi_._new_()
    self._nativeApi_._ctor_(self._handler_, 100)
    return self

  def __exit__(self, type, value, traceback):
    self._nativeApi_._dtor_(self._handler_, self._nativeApi_._freevalue_)
    self._nativeApi_._delete_(self._handler_)

  def size(self):
    return self._nativeApi_._size_(self._handler_) def push(self, item):
    result = self._nativeApi_._push_(self._handler_,
            self._nativeApi_._copyvalue_(item.encode('utf-8'), len(item)));
    if result != 1:
      raise Exception("Stack is full!")
...
```

Интеграция с Python

Пример на языке Python, который использует класс Stack (python/stack.py)

```
if __name__ == "__main__":
  with Stack() as stack:
    stack.push("Hello")
    stack.push("World")
    stack.push("!")
    print("Size after pushes:" + str(stack.size()))
    while stack.size() > 0:
      print(stack.pop())
    print("Size after pops:" + str(stack.size()))
    stack.push("Ba");
    stack.push("Bye!");
    print("Size before clear:" + str(stack.size()))
    stack.clear()
    print("Size after clear:" + str(stack.size()))
```

При входе в блок with вызывается функция __enter__, а обращение к экземпляру класса Stack происходит с помощью переменной stack. При выходе из блока with вызывается функция __exit__. Это дает нам возможность освободить соответствующие машинно-зависимые ресурсы (в данном случае объект стека C), когда они больше не нужны.

Интеграция с Python

run_python.sh

```
set -x
export CSTACK_LIB_PATH=..
LD_LIBRARY_PATH=$CSTACK_LIB_PATH python3 stack.py
```

Выполнение демонстрационного примера на Python

```
$ cd python
$ ./run_linux.sh
++ export CSTACK_LIB_PATH=..
++ CSTACK_LIB_PATH=..
++ LD_LIBRARY_PATH=..
++ python3 stack.py
Size after pushes:3
b'!'
b'World'
b'Hello'
Size after pops:0
Size before clear:2
Size after clear:0
```

Интеграция с Go

Язык программирования Golang (или просто Go) поддерживает простую интеграцию с машинно-зависимыми разделяемыми библиотеками. Он преподносится как системный язык, и его можно считать следующей итерацией C и C++. Поэтому мы ожидаем, что с загрузкой и использованием машинно-зависимых библиотек в Golang не будет никаких проблем.
В Golang для вызова любого кода на C и загрузки разделяемых объектных файлов предусмотрен встроенный пакет cgo.

Интеграция с Go

Класс Stack, который использует загруженный объектный файл libcstack.so (go/stack.go)

```
package main

import "C"
import (
  "fmt"
)

type Stack struct {
  handler *C.cstack_t
}

func NewStack() *Stack {
  s := new(Stack)
  s.handler = C.cstack_new()
  C.cstack_ctor(s.handler, 100)
  return s
}
 ...
```

Чтобы использовать пакет cgo, нужно импортировать C. Он загружает разделяемые библиотеки, указанные в псевдодирективе #cgo. С помощью директивы #cgo LDFLAGS: -L.. -lcstack мы указали, что нам нужно загрузить библиотеку libcstack.so. Обратите внимание: параметры CFLAGS и LDFLAGS определяют флаги, которые передаются непосредственно компилятору C и компоновщику соответственно.

Мы также указали путь, по которому нужно проводить поиск объектного файла. После этого можем использовать структуру C для вызова загруженных машиннозависимых функций. Например, с помощью C.cstack_new() мы вызвали соответствующую функцию из библиотеки для работы со стеком. Пакет cgo позволяет сделать это довольно легко.

Интеграция с Go

Пример, написанный на Go и использующий класс Stack (go/stack.go)

```go
func main() {
  var stack = NewStack()
  stack.Push("Hello")
  stack.Push("World")
  stack.Push("!")
  fmt.Println("Stack size:", stack.Size())
  for stack.Size() > 0 {
    _, str := stack.Pop()
    fmt.Println("Popped >", str)
  }
  stack.Destroy()
}
```

```
$ cd go
$ go build -o stack.out stack.go
$ LD_LIBRARY_PATH=$PWD/.. ./stack.out
Stack size: 3
Popped > !
Popped > World
Popped > Hello
```

Запуск примера на Go

Golang, в отличие от Python, программу нужно сначала скомпилировать и только потом запускать. Кроме того, мы должны установить переменную среды LD_LIBRARY_PATH, чтобы исполняемый файл мог найти и загрузить libcstack.so.

План лекции

Интеграция с другими языками

35 минут

Модульное тестирование и отладка

35 минут

Системы сборки

20 минут

Введение

Прежде чем передавать код клиентам, его необходимо тщательно протестировать, и неважно, какой язык программирования вы используете и какого рода приложение разрабатываете.

Написание тестов не является чем-то новым, и в любом современном программном проекте можно найти сотни и даже тысячи тестов. В наши дни писать тесты для ПО нужно обязательно, а доставка кода без надлежащей проверки крайне не приветствуется.

Мы обсудим средства и методы отладки, с помощью которых можно искать проблемы в программах на C. Тестирование и отладка всегда дополняли друг друга; когда тест проваливается, обычно далее следует анализ и отладка соответствующего кода.

Тестирование программного обеспечения

Тестирование ПО — большой и важный раздел программирования, имеющий собственную терминологию и множество концепций.

Когда речь заходит о тестировании программного обеспечения, сразу возникают вопросы: что мы тестируем и с какой целью? В целом мы тестируем отдельные аспекты системы. Они могут быть функциональными и нефункциональными. Иными словами, могут относиться как к определенным возможностям системы, так и к неким показателям, которые демонстрируются в момент предоставления этих возможностей.

Функциональное тестирование относится к определенным возможностям, входящим в список функциональных требований. Эти тесты предоставляют определенный ввод какому-то программному элементу, такому как функция, модуль, компонент или программная система, и ожидают получить от них определенный вывод. Тест считается пройденным только в случае получения ожидаемого вывода.

Нефункциональное тестирование относится к уровню качества, на котором программный элемент, такой как функция, модуль, компонент или система в целом, выполняет определенные функции. Эти тесты обычно измеряют различные показатели, например потребление памяти, время выполнения, конфликты при блокировках и уровень безопасности, и оценивают, насколько хорошо элемент справился со своими обязанностями. Тест считается пройденным, только если измеренный показатель находится в допустимом диапазоне. Ожидания относительно этих показателей выводятся из нефункциональных требований, предъявляемых к системе.

Уровни тестирования

В любой программной системе можно предусмотреть следующие уровни тестирования :

- модульное тестирование;
- интеграционное тестирование;
- системное тестирование;
- приемочное тестирование;
- регрессионное тестирование.

В модульном тестировании проверяется единица функциональности.

Компонент — часть программной системы с четко определенным набором функций; компоненты, собранные воедино, составляют всю систему в целом. Когда единицей функциональности выступает компонент, процесс тестирования называется ***компонентным***. На данном уровне можно проводить как функциональное, так и нефункциональное тестирование.

Модули в совокупности формируют компонент. В компонентном тестировании проверяются отдельные изолированные компоненты. Но если их сгруппировать, то возникнет необходимость в другом уровне тестирования, который относится к их функциональности или характеристикам, — в ***интеграционном тестировании***

Тестирование всей системы в целом происходит на другом уровне. Здесь у нас есть набор всех компонентов, которые полностью интегрированы. Таким образом, мы проверяем, соответствуют ли возможности, предоставляемые системой, и ее характеристики заявленным требованиям.

Уровни тестирования

Еще один уровень предусмотрен для проверки системы на соответствие бизнес-требованиям с точки зрения заинтересованной стороны или конечного пользователя. Это так называемое ***приемочное тестирование***. Как и системное, оно относится ко всей системе в целом, однако на самом деле оба уровня существенно отличаются. Вот лишь несколько аспектов разницы:

- за системное тестирование отвечают разработчики и тестировщики, а приемочное обычно проводят конечные пользователи или заинтересованная сторона;
- приемочное тестирование охватывает только функциональные требования, а системное — еще и нефункциональные;
- в системном тестировании в качестве ввода обычно используется небольшой, заранее подготовленный набор данных, тогда как приемочное имеет дело с настоящими данными, которые поступают в систему в режиме реального времени.

Прекрасное объяснение всех различий можно найти на странице.

Когда в программную систему вносятся изменения, необходимо убедиться в том, что функциональные и нефункциональные тесты по-прежнему актуальны. Для этого предусмотрен отдельный уровень, известный как ***регрессионное тестирование***. Его задача — подтвердить, что внесение изменений не привело к появлению регрессий. В рамках этого тестирования мы заново проводим все модульные, интеграционные и сквозные (системные) тесты, как функциональные, так и нефункциональные, и смотрим, провалились ли какие-либо из них.

Уровни тестирования

Еще один уровень предусмотрен для проверки системы на соответствие бизнес-требованиям с точки зрения заинтересованной стороны или конечного пользователя. Это так называемое ***приемочное тестирование***. Как и системное, оно относится ко всей системе в целом, однако на самом деле оба уровня существенно отличаются. Вот лишь несколько аспектов разницы:

- за системное тестирование отвечают разработчики и тестировщики, а приемочное обычно проводят конечные пользователи или заинтересованная сторона;
- приемочное тестирование охватывает только функциональные требования, а системное — еще и нефункциональные;
- в системном тестировании в качестве ввода обычно используется небольшой, заранее подготовленный набор данных, тогда как приемочное имеет дело с настоящими данными, которые поступают в систему в режиме реального времени.

Прекрасное объяснение всех различий можно найти на странице.

Когда в программную систему вносятся изменения, необходимо убедиться в том, что функциональные и нефункциональные тесты по-прежнему актуальны. Для этого предусмотрен отдельный уровень, известный как ***регрессионное тестирование***. Его задача — подтвердить, что внесение изменений не привело к появлению регрессий. В рамках этого тестирования мы заново проводим все модульные, интеграционные и сквозные (системные) тесты, как функциональные, так и нефункциональные, и смотрим, провалились ли какие-либо из них.

Модульное тестирование

Как уже объяснялось в предыдущем подразделе, в рамках модульного тестирования проверяются изолированные модули любых размеров — от функции до компонента. Это применимо как к C, так и к C++, только во втором случае роль модулей могут играть еще и классы. Самый важный аспект модульного тестирования состоит в том, что проверяемые модули должны быть изолированы друг от друга. Например, если одна функция зависит от другой, то мы должны как-то протестировать их по отдельности.

factorial.h и factorial.c

```
#ifndef _FACTORIAL_
#define _FACTORIAL_
#include <stdint.h>
#include <unistd.h>
typedef int64_t (*int64_feed_t)();
int64_t next_even_number();
int64_t calc_factorial(int64_feed_t feed);
#endif
```

calc_factorial принимает указатель на функцию, который возвращает целое число. С помощью этого указателя будут считываться целочисленные значения, для которых нужно вычислить факториал.

```
#include "factorial.h"
int64_t next_even_number() {
    static int feed = -2;
    feed += 2;
    if (feed >= 10)
        feed = 0;
    return feed;
}
int64_t calc_factorial(int64_feed_t feed) {
    int64_t fact = 1;
    int64_t number = feed();
    for (int64_t i = 1; i <= number; i++)
        fact *= i;
    return fact;
}
```

Модульное тестирование

Пример 33_1 main.c

```c
#include <stdio.h>
#include "factorial.h"
int main(int argc, char **argv) {
    for (size_t i = 1; i <= 12; i++) {
        printf("%lu\n", calc_factorial(next_even_number));
    }
    return 0;
}
```

Сборка и запуск примера

```
$ gcc -c factorial.c -o impl.o
$ gcc -c main.c -o main.o
$ gcc impl.o main.o -o a.out
$ ./a.out
1
2
24
720
40320
1
2
24
720
40320
1
2
```

В данном примере у нас две функции (не считая main), next_ even_number и calc_factorial, поэтому нам нужно по отдельности протестировать два разных изолированных модуля.

Модульное тестирование

Пример 33_1 next_even_number__tests.c

```c
#include <assert.h>
#include "factorial.h"
void TESTCASE_next_even_number__even_numbers_should_be_returned() {
    assert(next_even_number() == 0);
    assert(next_even_number() == 2);
    assert(next_even_number() == 4);
    assert(next_even_number() == 6);
    assert(next_even_number() == 8);
}
void TESTCASE_next_even_number__numbers_should_rotate() {
    int64_t number = next_even_number();
    next_even_number();
    next_even_number();
    next_even_number();
    next_even_number();
    int64_t number2 = next_even_number();
    assert(number == number2);
}
```

Что же такое тестовый случай? В отдельно взятом модуле есть разные случаи, которые можно протестировать. Простейший пример — передача модулю разных входных значений с последующей проверкой того, соответствует ли вывод ожиданиям. В нашем примере мы можем передать функции calc_factorial значение 0, ожидая получить 1 на выходе. Того же результата можно ожидать при передаче -1.

Каждый из этих вариантов может служить тестовым случаем. Группа тестовых случаев называется тестовым набором. Элементы одного тестового набора могут относиться к разным модулям.

Модульное тестирование

Пример 33_1 factorial__tests.c

```
#include <assert.h>
#include "factorial.h"
int64_t input_value = -1;
int64_t feed_stub() { return input_value; }
void TESTCASE_calc_factorial__fact_of_zero_is_one() {
    input_value = 0;
    int64_t fact = calc_factorial(feed_stub);
    assert(fact == 1);
}
void TESTCASE_calc_factorial__fact_of_negative_is_one() {
    input_value = -10;
    int64_t fact = calc_factorial(feed_stub);
    assert(fact == 1);
}
void TESTCASE_calc_factorial__fact_of_5_is_120() {
    input_value = 5;
    int64_t fact = calc_factorial(feed_stub);
    assert(fact == 120);
}
```

Итак, мы определили три тестовых случая для функции calc_factorial. Обратите внимание на функцию feed_stub. Она соблюдает тот же контракт, что и next_even_ number, но при этом имеет очень простое определение. Она всего лишь возвращает значение, хранящееся в статической переменной input_value. Тестовые случаи могут инициализировать эту переменную перед вызовом calc_ facorial.

Используя упомянутую выше заглушку, мы можем изолировать функцию calc_ facorial и протестировать ее отдельно. Тот же подход применим и в объектноориентированных языках программирования, таких как C++ или Java, но там в качестве заглушек нужно определять классы и объекты.

Модульное тестирование

Пример 33_1 Средство выполнения тестов tests.c

```c
#include <stdio.h>
void TESTCASE_next_even_number__even_numbers_should_be_returned();
void TESTCASE_next_even_number__numbers_should_rotate();
void TESTCASE_calc_factorial__fact_of_zero_is_one();
void TESTCASE_calc_factorial__fact_of_negative_is_one();
void TESTCASE_calc_factorial__fact_of_5_is_120();
int main(int argc, char **argv) {
    TESTCASE_next_even_number__even_numbers_should_be_returned();
    TESTCASE_next_even_number__numbers_should_rotate();
    TESTCASE_calc_factorial__fact_of_zero_is_one();
    TESTCASE_calc_factorial__fact_of_negative_is_one();
    TESTCASE_calc_factorial__fact_of_5_is_120();
    printf("All tests are run successfully.\n");
    return 0;
}
```

В языке С заглушка — это определение функции, соответствующее объявлению, которое используется в логике тестируемого модуля. Что еще важнее, заглушка должна быть простой и возвращать значение, которое будет использоваться исключительно в тестовом случае.

В C++ заглушкой может быть как определение функции, которое соответствует ее объявлению, так и класс, который реализует интерфейс. В других объектно-ориентированных языках, где нет самостоятельных функций (таких как Java), возможен только второй вариант, где заглушка представляет собой объект, созданный из класса-заглушки. У заглушки должно быть простое определение, которое подходит лишь для тестов, а не реального использования.

Модульное тестирование

Пример 33_1 Сборка и запуск средства выполнения тестов

```
$ gcc -g -c factorial.c -o impl.o
$ gcc -g -c next_even_number__tests.c -o tests1.o
$ gcc -g -c calc_factorial__tests.c -o tests2.o
$ gcc -g -c test.c -o main.o
$ gcc impl.o tests1.o tests2.o main.o -o tests.out
$ ./tests.out
All tests are run successfully.
$ echo $?
0
```

Данный код возвращает 0, только если все тестовые случаи внутри функций main выполняются успешно. Обратите внимание на параметр -g, который добавляет отладочные символы в итоговый исполняемый файл. Тесты чаще всего собираются в отладочном режиме, поскольку в случае провала одного из них нам сразу же нужна точная трассировка стека и дополнительная отладочная информация, чтобы продолжить наше расследование. Более того, инструкции assert обычно удаляются из сборки выпуска, но в исполняемом файле средства выполнения тестов должны присутствовать.

Как видите, все тесты успешно пройдены. Мы также можем проверить код выхода у процесса средства выполнения тестов, используя команду echo $?, и посмотреть, возвращает ли она 0.

Модульное тестирование

Пример 33_1 Функция calc_factorial с измененной сигнатурой, которая не принимает указатель на функцию factorial_2.c

```
...
int64_t calc_factorial() {
    int64_t fact = 1;
    int64_t number = next_even_number();
    for (int64_t i = 1; i <= number; i++)
        fact *= i;
    return fact;
}
```

Данный код не такой тестируемый. Мы не можем протестировать функцию calc_factorial, не вызывая next_even_number, — то есть не внося «грязных» изменений в определение символа next_even_number в итоговом исполняемом файле.

На самом деле обе версии calc_factorial делают одно и то же, но первоначальный лучше поддается тестированию, поскольку его можно проверять в изоляции. Написание тестируемого кода — непростая задача, которая всегда требует предусмотрительности и дополнительных усилий. Есть разные мнения о том, сколько именно накладных расходов для этого нужно, но неизбежность этих расходов не вызывает сомнений. Однако выгода очевидна. Вы не уследите за модулем, не имеющим тестов, в который со временем вносятся все новые изменения.

Тестовые дублеры

Объекты, которые пытаются имитировать зависимости модуля, называются тестовыми дублерами. Существует две разновидности: макеты и фиктивные заполнители. ***Макетные функции*** (или макетные объекты в целом, если говорить об объектноориентированных языках) можно модифицировать, указывая вывод для определенного ввода. Таким образом, перед запуском логики теста мы указываем то, что должно вернуться из макетной функции при определенном вводе, и в ходе выполнения эта функция будет вести себя так, как было определено заранее. ***Макетные объекты*** в целом могут иметь определенные ожидания, но могут и выполнять нужные нам утверждения. Ожидания для макетных объектов задаются перед запуском теста.

Чтобы получить в тесте упрощенную версию настоящей и, возможно, сложной функциональности, используется ***фиктивная функция***. Например, вместо реальной файловой системы можно задействовать некое упрощенное хранилище в памяти. В компонентном тестировании, к примеру, фиктивные реализации позволяют заменить другие компоненты, имеющие сложную функциональность.

Покрытие кода: Теоретически все модули должны иметь соответствующие тестовые наборы, и каждый набор должен содержать все тестовые случаи, которые проходятся по всем возможным ветвям кода. Но это теория. А на практике тесты предусмотрены лишь для какой-то части модулей. Наши тестовые случаи обычно не покрывают все имеющиеся ответвления. Доля модулей, имеющих подходящие тестовые случаи, называется покрытием кода или охватом тестирования. Чем она больше, тем выше вероятность того, что вы получите уведомления о нежелательных изменениях.

Компонентное тестирование

```c
#include <assert.h>
#include "factorial.h"
void TESTCASE_component_test__factorials_from_0_to_8() {
    assert(calc_factorial(next_even_number) == 1);
    assert(calc_factorial(next_even_number) == 2);
    assert(calc_factorial(next_even_number) == 24);
    assert(calc_factorial(next_even_number) == 720);
    assert(calc_factorial(next_even_number) == 40320);
}
void TESTCASE_component_test__factorials_should_rotate() {
    int64_t number = calc_factorial(next_even_number);
    for (size_t i = 1; i <= 4; i++)
        calc_factorial(next_even_number);
    int64_t number2 = calc_factorial(next_even_number);
    assert(number == number2);
}
int main(int argc, char **argv){
    TESTCASE_component_test__factorials_from_0_to_8();
    TESTCASE_component_test__factorials_should_rotate();
    return 0;
}
```

Компонентное тестирование — частный случай модульного тестирования. **Пример 33_1** гипотетический компонент в рамках примера component_tests.c и поместим в него две функции, которые у нас уже есть. Обратите внимание: из компонента обычно получается исполняемый файл или библиотека. Представим, что наш компонент будет собираться в библиотеку с двумя функциями.

У нас должна быть возможность протестировать функциональность компонента. Мы по-прежнему будем писать тестовые случаи, но они будут отличаться изолируемыми модулями. Раньше это были функции, а теперь мы собираемся изолировать компонент, состоящий из двух функций, которые работают сообща.

Библиотеки тестирования для C

Для интеграционного тестирования можно выбрать другой язык программирования. В целом интеграционное и системное тестирование отличается высокой сложностью, поэтому нам нужно использовать какие-то фреймворки для автоматизации тестирования, чтобы упростить написание тестов и выполнять их без лишних усилий. В рамках данной автоматизации будет использоваться предметно-ориентированный язык (domain-specific language, DSL); он упростит написание и выполнение тестов. Для этой цели подходит много разных языков, однако скриптовые, такие как Unix shell, Python, JavaScript и Ruby, пользуются наибольшей популярностью. В автоматизации тестов активно применяются и другие языки программирования, включая Java.

Ниже перечислены одни из самых известных фреймворков модульного тестирования:

- Check (от автора приведенной выше ссылки);
- AceUnit;
- GNU Autounit;
- cUnit;
- CUnit;
- CppUnit;
- CuTest;
- embUnit;
- MinUnit;
- Google Test;
- CMocka.

CMocka

Первое преимущество CMocka, на которое следует обратить внимание, таково: этот фреймворк написан исключительно на C и зависит только от стандартной библиотеки данного языка — никаких других библиотек он не использует. Поэтому тесты можно собирать компилятором C, что делает тестовую среду очень близкой к реальной. Фреймворк CMocka доступен для многих платформ, включая macOS, Linux и даже Microsoft Windows.

CMocka де-факто стандартный фреймворк для модульного тестирования в C. Он поддерживает средства тестирования (test fixtures), которые позволяют инициализировать и очищать тестовую среду перед выполнением тестового случая и после него. Вдобавок CMocka поддерживает макетирование функций, что очень полезно при написании макета какой-либо функции на C.

В CMocka каждый тестовый случай должен возвращать void и принимать аргумент void**. Аргумент-указатель будет использоваться для получения информации, state, которая относится к отдельно взятому тестовому случаю. В функции main мы создадим список тестовых случаев и в конце вызовем cmocka_run_group_tests, чтобы выполнить все модульные тесты.

Помимо самих тестовых случаев, мы видим две новые функции: setup и tear_down. Это средства тестирования. Они вызываются перед выполнением каждого тестового случая и после него и предназначены для его подготовки и уничтожения. Первое вызывается перед тестовым случаем, а второе — после него. Их можно было бы назвать как угодно, но для ясности были выбраны имена setup и tear_down.

CMocka

Пример 33_1 Тестовые случаи на основе CMocka для примера (cmocka_tests.c)

```
#include <cmocka.h>
int64_t input_value = -1;
int64_t feed_stub() { return input_value; }
void calc_factorial__fact_of_zero_is_one(void **state) {
    input_value = 0;
    int64_t fact = calc_factorial(feed_stub);
    assert_int_equal(fact, 1);
}
...
void next_even_number__even_numbers_should_be_returned(void **state) {
    assert_int_equal(next_even_number(), 0);
    assert_int_equal(next_even_number(), 2);
...
}
int main(int argc, char **argv) {
    const struct CMUnitTest tests[] = {
        cmocka_unit_test(calc_factorial__fact_of_zero_is_one),
        ...
    };
    return cmocka_run_group_tests(tests, setup, tear_down);
}
```

CMocka

Еще одно различие между тестовыми случаями, написанными с помощью CMocka, и теми, которые мы видели в предыдущих разделах, заключается в использовании других функций-утверждений. Библиотеки тестирования предоставляют широкое разнообразие утверждений, позволяющих получить дополнительные сведения о проваленном тесте, в то время как стандартная функция assert немедленно завершает программу, не предоставляя подробной информации. В представленном выше коде используется функция assert_int_equal, которая проверяет, равны ли два целых числа.

```
$ gcc -g -c cmocka_tests.c -o cmocka_tests.o
$ gcc impl.o cmocka_tests.o -lcmocka -o cmocka_tests.out
$ ./cmocka_tests.out
[==========] Running 5 test(s).
[ RUN      ] calc_factorial__fact_of_zero_is_one
[       OK ] calc_factorial__fact_of_zero_is_one
[ RUN      ] calc_factorial__fact_of_negative_is_one
[       OK ] calc_factorial__fact_of_negative_is_one
[ RUN      ] calc_factorial__fact_of_5_is_120
[       OK ] calc_factorial__fact_of_5_is_120
[ RUN      ] next_even_number__even_numbers_should_be_returned
[       OK ] next_even_number__even_numbers_should_be_returned
[ RUN      ] next_even_number__numbers_should_rotate
[       OK ] next_even_number__numbers_should_rotate
[==========] 5 test(s) run.
[  PASSED  ] 5 test(s).
```

Чтобы скомпилировать данную программу, нужно сначала установить CMocka. В системах Linux на основе Debian для этого достаточно выполнить sudo apt-get install libcmocka-dev, а в macOS можно воспользоваться командой brew install cmocka. В Интернете можно найти множество справочных материалов, которые помогут вам с процессом установки.

CMocka

Изменение одного из тестовых случаев на основе CMocka

```
void next_even_number__even_numbers_should_be_returned(void **state)
{
    assert_int_equal(next_even_number(), 1);
    ...
}
```

```
$ gcc -g -c cmocka_tests.c -o cmocka_tests.o
$ gcc impl.o
$ ./cmocka_tests.out
[==========] Running 5 test(s).
[ RUN      ] calc_factorial__fact_of_zero_is_one
...
[       OK ] calc_factorial__fact_of_5_is_120
[ RUN      ] next_even_number__even_numbers_should_be_returned
[  ERROR   ] --- 0 != 0x1
[   LINE   ] --- cmocka_tests.c:29: error: Failure!
[  FAILED  ] next_even_number__even_numbers_should_be_returned
[ RUN      ] next_even_number__numbers_should_rotate
[       OK ] next_even_number__numbers_should_rotate
[==========] 5 test(s) run.
[  PASSED  ] 4 test(s).
[  FAILED  ] 1 test(s), listed below:
[  FAILED  ] next_even_number__even_numbers_should_be_returned
 1 FAILED TEST(S)
```

Один из тестовых случаев провалился, и причина указана в сообщении об ошибке среди журнальных записей. Не была пройдена проверка равенства целых чисел. Применение функции assert_int_equal вместо обычного вызова assert позволяет CMocka выводить в журнале выполнения полезную информацию, не ограничиваясь одной остановкой программы.

CMocka

Заголовочный файл в примере 31_1 (random.h)

```
#ifndef _RANDOM_
#define _RANDOM_
#define TRUE 1
#define FALSE 0
typedef int bool_t;
bool_t random_boolean();
#endif
```

Определение функции random_boolean в примере 31_1 (random.c)

```
#include <stdlib.h>
#include <stdio.h>
#include "random.h"
bool_t random_boolean() {
    int number = rand();
    return (number % 2);
}
```

В примере демонстрируется макетирование функций с помощью CMocka. Этот фреймворк позволяет создать макет функции, который при получении определенного ввода будет возвращать заданный результат.

Мы возьмем стандартную функцию rand для генерирования случайных чисел. У нас также есть функция random_boolean, которая возвращает булево значение в зависимости от того, какое число мы получили из rand: четное или нечетное.

Прежде всего нужно сказать, что мы не можем позволить random_boolean использовать настоящее определение функции rand, поскольку, как можно догадаться по ее имени, она генерирует случайные числа, а элемент случайности в наших тестах недопустим. Тесты проверяют результат на соответствие ожиданиям, поэтому предоставляемый ввод должен быть предсказуемым. Более того, определение rand — часть стандартной библиотеки C, такой как glibc в Linux, и его не получится легко заменить заглушкой.

Один за простейших способов подмены определения rand в C заключается в правке символов в итоговом объектном файле. В таблице символов объектного файла можно найти запись для rand, которая ссылается на соответствующее определение в стандартной библиотеке C. Если изменить эту запись так, чтобы она указывала на другое определение функции rand в наших тестовых двоичных файлах, то мы сможем легко подставить вместо rand нашу заглушку.

CMocka

Пример 33_1 Написание тестовых случаев CMocka с использованием функции-заглушки (cmocka_tests_with_stub.c)

```
#include <stdlib.h>
// Нужно для библиотеки CMocka
#include <stdarg.h> #include <stddef.h> #include <setjmp.h> #include <cmocka.h>
#include "random.h"
int next_random_num = 0;
int __wrap_rand() { return next_random_num; }
void test_even_random_number(void **state) {
    next_random_num = 10;
    assert_false(random_boolean());
}
void test_odd_random_number(void **state) {
    next_random_num = 13;
    assert_true(random_boolean());
}
int main(int argc, char **argv) {
    const struct CMUnitTest tests[] = {cmocka_unit_test(test_even_random_number),
cmocka_unit_test(test_odd_random_number)};
    return cmocka_run_group_tests(tests, NULL, NULL);
}
```

CMocka

Пример 33_1 Сборка и запуск модульных тестов на основе CMocka для примера

```
$ cd rand
$ gcc -g -c random.c -o impl.o
$ gcc -g -c cmocka_tests_with_stub.c -o tests.o
$ gcc impl.o tests.o -lcmocka -o cmocka_tests_with_stub.out
$ ./cmocka_tests_with_stub.out
[==========] Running 2 test(s).
[ RUN      ] test_even_random_number
[  ERROR   ] --- random_boolean()
[   LINE   ] --- cmocka_tests_with_stub.c:13: error: Failure!
[  FAILED  ] test_even_random_number
[ RUN      ] test_odd_random_number
[  ERROR   ] --- random_boolean()
[   LINE   ] --- cmocka_tests_with_stub.c:18: error: Failure!
[  FAILED  ] test_odd_random_number
[==========] 2 test(s) run.
[  PASSED  ] 0 test(s).
[  FAILED  ] 2 test(s), listed below:
[  FAILED  ] test_even_random_number
[  FAILED  ] test_odd_random_number

 2 FAILED TEST(S)
```

CMocka

Пример 33_1 Сборка и запуск модульных тестов на основе CMocka для примера 2.2 после создания обертки для символа rand

```
$ gcc impl.o tests.o -lcmocka -Wl,--wrap=rand -o cmocka_tests_with_stub.out
$ ./cmocka_tests_with_stub.out
[==========] Running 2 test(s).
[ RUN      ] test_even_random_number
[       OK ] test_even_random_number
[ RUN      ] test_odd_random_number
[       OK ] test_odd_random_number
[==========] 2 test(s) run.
[  PASSED  ] 2 test(s).
```

Как видно в выводе, стандартная функция rand больше не вызывается, а вместо нее наша заглушка возвращает заданное нами значение. Чтобы функция __wrap_rand вызывалась вместо rand, при компоновке с помощью gcc использовался параметр -Wl,--wrap=rand.

Этот параметр доступен только в программе ld в Linux, поэтому для подмены функций в macOS или других системах, в которых нет компоновщика GNU, следует использовать другие приемы, такие как взаимное позиционирование.

Параметр --wrap=rand заставляет компоновщик обновить запись с символом rand в таблице символов итогового исполняемого файла, чтобы она ссылалась на определение функции __wrap_rand.

CMocka

Пример 33_1 Написание тестовых случаев на основе CMocka с использованием макетной функции (cmocka_tests_with_mock.c)

```
#include <stdlib.h>
// Нужно для библиотеки CMocka
#include <stdarg.h> #include <stddef.h> #include <setjmp.h> #include <cmocka.h>
#include "random.h"
int __wrap_rand() { return mock_type(int); }
void test_even_random_number(void **state) {
    will_return(__wrap_rand, 10);
    assert_false(random_boolean());
}
void test_odd_random_number(void **state) {
    will_return(__wrap_rand, 13);
    assert_true(random_boolean());
}
int main(int argc, char **argv) {
    const struct CMUnitTest tests[] = {cmocka_unit_test(test_even_random_number),
cmocka_unit_test(test_odd_random_number)};
    return cmocka_run_group_tests(tests, NULL, NULL);
}
```

Теперь, зная о вызове функции-обертки __wrap_rand, мы можем объяснить код, относящийся к макетированию. Макет создается с помощью пары функций will_ return и mock_type. Сначала вызывается функция will_return; она определяет значение, которое должен вернуть макет. Затем, когда вызывается макет (в данном случае __wrap_rand), функция mock_type возвращает заданное значение.

CMocka

Пример 33_1 Написание тестовых случаев на основе CMocka с использованием макетной функции (cmocka_tests_with_mock.c)

```
#include <stdlib.h>
// Нужно для библиотеки CMocka
#include <stdarg.h> #include <stddef.h> #include <setjmp.h> #include <cmocka.h>
#include "random.h"
int __wrap_rand() { return mock_type(int); }
void test_even_random_number(void **state) {
    will_return(__wrap_rand, 10);
    assert_false(random_boolean());
}
void test_odd_random_number(void **state) {
    will_return(__wrap_rand, 13);
    assert_true(random_boolean());
}
int main(int argc, char **argv) {
    const struct CMUnitTest tests[] = {cmocka_unit_test(test_even_random_number),
cmocka_unit_test(test_odd_random_number)};
    return cmocka_run_group_tests(tests, NULL, NULL);
}
```

Теперь, зная о вызове функции-обертки __wrap_rand, мы можем объяснить код, относящийся к макетированию. Макет создается с помощью пары функций will_ return и mock_type. Сначала вызывается функция will_return; она определяет значение, которое должен вернуть макет. Затем, когда вызывается макет (в данном случае wrap rand), функция mock type возвращает заданное значение.

Google Test

Google Test — фреймворк, пригодный для модульного тестирования программ на C и C++. Он написан на языке C++, но подходит и для тестирования кода на C. Некоторые считают, что так делать не стоит, поскольку в этом случае тестовая и реальная среды используют разные компиляторы и компоновщики.

Пример 33_1 Измененный заголовочный файл в примере (factorial_plus.h)

```
#ifndef _FACTORIAL_PLUS_H_
#define _FACTORIAL_PLUS_H_
#include <stdint.h>
#include <unistd.h>
#if __cplusplus
extern "C"
{
#endif
    typedef int64_t (*int64_feed_t)();
    int64_t next_even_number();
    int64_t calc_factorial(int64_feed_t feed);
#if __cplusplus
}
#endif
#endif
```

Мы разместили определение в блоке extern C { ... }, который остается в программе только при наличии макроса _cplusplus. В результате этого изменения, в случае использования компилятора C++, символы в итоговых объектных файлах не должны декорироваться. Иначе, когда компоновщик попытается найти определение декорированных символов, произойдет ошибка компоновки. Если вы незнакомы с декорированием имен в C++

Google Test

Пример 33_1 Тесты написанные с помощью Google Test

```
// Нужно для библиотеки Google Test
#include <gtest/gtest.h>
#include "factorial_plus.h"
int64_t input_value = -1;
int64_t feed_stub() { return input_value; }
TEST(calc_factorial, fact_of_zero_is_one)
{
    input_value = 0;
    int64_t fact = calc_factorial(feed_stub);
    ASSERT_EQ(fact, 1);
}
...
TEST(next_even_number, even_numbers_should_be_returned
{
    ASSERT_EQ(next_even_number(), 0);
    ASSERT_EQ(next_even_number(), 2);
    ASSERT_EQ(next_even_number(), 4);
    ASSERT_EQ(next_even_number(), 6);
    ASSERT_EQ(next_even_number(), 8);
}
int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Тестовые случаи определены с помощью макроса TEST(...). Это пример того, насколько легко макросы позволяют сформировать DSL. Существуют и другие макросы, такие как TEST_F(...) и TEST_P(...), но они предназначены для C++. Первый аргумент макроса представляет собой имя тестового класса (фреймворк Google Test написан для объектно-ориентированного кода на C++), который можно считать набором тестов, содержащим ряд тестовых случаев. Второй аргумент — имя тестового случая.

Обратите внимание: макрос ASSERT_EQ позволяет делать утверждения о равенстве объектов, а не только целых чисел. Google Test предоставляет большое количество макросов для проверки ожиданий, что делает этот проект полноценным фреймворком для модульного тестирования.

Google Test

Пример 33_1 Сборка и запуск модульных тестов Google Test

```
$ gcc -g -c factorial_plus.c -o impl_plus.o
$ g++ -std=c++11 -g -c gtests.cpp -o gtests.o
$ g++ impl_plus.o gtests.o -lgtest -lpthread -o gtests.out
$ ./gtests.out
[==========] Running 5 tests from 2 test suites.
[----------] Global test environment set-up.
[----------] 3 tests from calc_factorial
[ RUN      ] calc_factorial.fact_of_zero_is_one
[       OK ] calc_factorial.fact_of_zero_is_one (0 ms)
[ RUN      ] calc_factorial.fact_of_negative_is_one
[       OK ] calc_factorial.fact_of_negative_is_one (0 ms)
[ RUN      ] calc_factorial.fact_of_5_is_120
[       OK ] calc_factorial.fact_of_5_is_120 (0 ms)
[----------] 3 tests from calc_factorial (0 ms total)

[----------] 2 tests from next_even_number
[ RUN      ] next_even_number.even_numbers_should_be_returned
[       OK ] next_even_number.even_numbers_should_be_returned (0 ms)
[ RUN      ] next_even_number.numbers_should_rotate
[       OK ] next_even_number.numbers_should_rotate (0 ms)
[----------] 2 tests from next_even_number (0 ms total)

[----------] Global test environment tear-down
[==========] 5 tests from 2 test suites ran. (0 ms total)
[  PASSED  ] 5 tests.
```

Google Test

Изменение одного из тестовых случаев, написанных с использованием Google Test

Пример 33_1 Сборка и запуск модульных тестов Google Test

```
[==========] Running 5 tests from 2 test suites.
[----------] Global test environment set-up.
[----------] 3 tests from calc_factorial
[ RUN      ] calc_factorial.fact_of_zero_is_one
...
[       OK ] calc_factorial.fact_of_5_is_120 (0 ms)
[----------] 3 tests from calc_factorial (0 ms total)
[----------] 2 tests from next_even_number
[ RUN      ] next_even_number.even_numbers_should_be_returned
gtests.cpp:26: Failure
Expected equality of these values:
  next_even_number()
    Which is: 0
  1
[  FAILED  ] next_even_number.even_numbers_should_be_returned (0 ms)
[ RUN      ] next_even_number.numbers_should_rotate
[       OK ] next_even_number.numbers_should_rotate (0 ms)
[----------] 2 tests from next_even_number (0 ms total)
[----------] Global test environment tear-down
[==========] 5 tests from 2 test suites ran. (0 ms total)
[  PASSED  ] 4 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] next_even_number.even_numbers_should_be_returned

 1 FAILED TEST
```

```
TEST(next_even_number,
even_numbers_should_be_returned)
{
    ASSERT_EQ(next_even_number(), 1);
...
}
```

Отладка

Что такое отладка ПО? Бытует мнение, будто термин ***debug*** (отладка) появился во времена, когда компьютеры были такими большими, что в системное оборудование могли забираться настоящие ***насекомые*** (bugs) типа мотыльков, вызывая неполадки. Обязанностью некоторых работников ***с официальной должностью*** debuggers (дословно «борцы с насекомыми» — то, что мы называем отладчиками) было вылавливать насекомых из оборудования в аппаратном зале. Более подробно об этом можно почитать на странице.

Отладка — это расследование, целью которого является определение причины наблюдаемого дефекта путем анализа внутренностей или внешнего поведения программы. Обычно, запуская программу, мы рассматриваем ее в качестве некоего черного ящика. Но когда с результатами что-то не так или работа программы почему-то прерывается, нам нужно заглянуть поглубже и выяснить, откуда берется данная проблема. Это значит, что программу следует анализировать по принципу белого ящика, в котором все видно.

Вот почему у нас может быть две разные сборки программы: сборка выпуска и отладочная сборка. В сборке выпуска акцент на работе и функциональности, а программа в основном рассматривается в качестве черного ящика; в отладочных сборках мы можем отслеживать любые происходящие события и анализировать программу по принципу белого ящика. Отладочные сборки в основном предназначены для разработки и тестирования, а сборки выпуска используются для развертывания и работы в реальных условиях.

Отладка

Чтобы получить отладочную сборку, все элементы программного проекта или какая-то их часть должны содержать отладочные символы, которые позволяют разработчику выполнять трассировку стека программы и отслеживать ее поток выполнения. Обычно продукты сборки выпуска (исполняемый файл или библиотеки) не подходят для отладки — они недостаточно прозрачные для того, чтобы внешний наблюдатель мог исследовать внутренности программы.

Для отладки программ, как правило, используются отладчики. ***Отладчик*** — это самостоятельный инструмент, который подключается к заданному процессу, чтобы управлять им или отслеживать его работу. ***Отладчики*** — основное средство исследования проблем, но есть и другие инструменты, которые можно использовать для анализа памяти, конкурентных потоков выполнения или производительности программы. Мы поговорим о них чуть позже.

Существенная часть программных ошибок подлежит воспроизведению, но есть и такие ошибки, которые нельзя воспроизвести или наблюдать во время сеанса отладки. Это в основном связано с эффектом наблюдателя, согласно которому при попытке заглянуть внутрь программы вы влияете на то, как она работает, и можете предотвратить появление ряда дефектов. Такого рода проблемы являются катастрофическими, и их зачастую очень сложно исправить, поскольку вы не можете использовать средства отладки для поиска их причин!

К этой категории можно отнести некоторые ошибки с управлением потоками выполнения в высокопроизводительных средах.

Категории программных ошибок

За годы эксплуатации программного продукта можно получить тысячи отчетов об ошибках. Но категорий, на которые можно разделить эти ошибки, не так уж много. Виды программных дефектов:

Логические ошибки. Для исследования таких ошибок вам необходимо ориентироваться в коде и его потоках выполнения. Чтобы увидеть поток выполнения программы, к запущенному процессу нужно подключить отладчик. Только после этого данный поток можно будет трассировать и анализировать. При отладке программы можно также использовать журнальные записи работы, особенно если в итоговых двоичных файлах нет отладочных символов или если отладчик нельзя подключить к активному экземпляру программы.

Ошибки памяти. Связаны с памятью. Обычно их вызывают висячие указатели, переполнения буфера, двойное освобождение и т. д. Эти ошибки необходимо исследовать с помощью профилировщика памяти, который играет роль средства отладки для мониторинга и наблюдения за памятью.

Ошибки конкурентности. Многопроцессным и многопоточным программам всегда были свойственны одни из самых трудно выявляемых ошибок в компьютерной промышленности. Для обнаружения особенно коварных проблем, таких как состояние гонки или гонка данных, требуются специальные инструменты наподобие средств отладки потоков (thread sanitizers).

Проблемы с производительностью. Обновление программного обеспечения может привести к ухудшению производительности. Такие проблемы нужно исследовать за счет дополнительного тестирования или даже отладки. Журнальные записи выполнения, могут помочь в поиске изменений, которые вызвали ухудшения производительности.

Отладчики

Распространенные возможности, доступные в большинстве отладчиков.

- Отладчик, как и любая другая программа, выполняется в виде процесса. Процесс отладчика может присоединиться к другому процессу с заданным идентификатором.
- После успешного присоединения отладчик может управлять выполнением инструкций в заданном процессе.
- Отладчики могут заглядывать внутрь защищенной памяти процесса. Они также способны изменять ее содержимое.
- Почти все известные отладчики могут отследить исходный код, который относится к той или иной инструкции; для этого лишь требуется, чтобы переносимые объектные файлы содержали отладочные символы.
- Если в заданном объектном файле нет отладочных символов, то отладчик может показать дизассемблированный код соответствующей инструкции, что тоже может быть полезно.
- Некоторые отладчики предназначены только для определенных языков, но большинство из них универсальны. Языки на основе виртуальной машины Java (Java Virtual Machine, JVM), такие как Java, Scala и Groovy, должны использовать отладчики JVM, позволяющие просматривать и управлять внутренностями экземпляра JVM.
- У интерпретируемых языков, таких как Python, тоже есть свои отладчики, которые позволяют останавливать скрипты и управлять их выполнением.

Примеры отладчиков: Advanced Debugger (adb), GNU Debugger (gdb), Python, Java Platform Debugger Architecture (JPDA), OllyDbg, Microsoft Visual Studio Debugger

Средства проверки памяти

От средства проверки памяти обычно можно ожидать таких возможностей, как:

- подсчет общего объема выделенной, освобожденной и задействованной статической памяти, а также количества выделений кучи, стека и т. д.;
- обнаружение утечек памяти, что можно считать важнейшей функцией про-филировщиков;
- обнаружение некорректных операций чтения/записи, таких как выход за пределы буферов или массивов, запись на уже освобожденный участок памяти и т. д.;
- обнаружение проблемы двойного освобождения. Она возникает, когда программа пытается освободить область памяти, которая уже подверглась этой операции.

Средства проверки памяти делают одно и то же, но внутренние механизмы мониторинга операций с памятью, разделим их в зависимости от подходов, которые в них применяются.

1. ***Переопределение на этапе компиляции.*** Преимущество данного подхода в том, что он меньше всего влияет на производительность программы по сравнению с другими методиками. Но у него есть и недостаток: двоичные файлы приходится перекомпилировать. **Примеры:** LLVM AddressSanitizer (ASan), Memwatch, Dmalloc и Mtrace.
2. ***Переопределение на этапе компоновки.*** Данный вид средств проверки памяти похож на предыдущий, но отличается тем, что вам двоичные файлы достаточно скомпоновать с предоставленной библиотекой. Для проверки памяти на этапе компоновки можно использовать утилиту heap checker и gperftools.
3. ***Перехват на этапе выполнения.*** Средство проверки памяти на основе этого подхода выступает медиатором между программой и ОС, пытаясь перехватывать и отслеживать все операции работы с памятью и сообщать о любом некорректном поведении или обращении не по тому адресу. Оно также может создавать отчеты об утечках. Для перехвата на этапе выполнения можно использовать утилиту Memcheck, входящую в состав Valgrind.
4. ***Предварительная загрузка библиотек.*** Некоторые профилировщики применяют взаимное позиционирова для создания оберток вокруг стандартных функций для работы с памятью.

Средства отладки потоков

Средства отладки потоков — это программы, которые позволяют отлаживать многопоточные процессы и находить в них проблемы, связанные с конкурентностью. Вот лишь несколько проблем, которые они способны выявить:

- гонки данных и точное местонахождение операций чтения/записи, в которых они возникают. Эти операции могут находиться в разных потоках;
- неправильное использование API для работы с потоками, особенно того, который входит в стандарт POSIX и применяется в POSIX-совместимых системах;
- потенциальная взаимная блокировка;
- проблемы с порядком блокировки.

В следующем списке представлен ряд широко известных средств отладки потоков.

- Helgrind (от Valgrind) — еще один инструмент из состава Valgrind, в основном применяемый для отладки потоков. Кроме того, следует упомянуть о DRD — аналогичном средстве, которое тоже является частью Valgrind. Списки возможностей Helgrind и DRD. Для его запуска необходимо выполнить команду valgrind --tool=helgrind [path-to-executable].
- Intel Inspector — данный преемник Intel Thread Checker проводит анализ ошибок потоков выполнения и проблем с памятью. Поэтому его можно считать как отладчиком потоков, так и профилировщиком памяти.
- LLVM ThreadSanitizer (TSan) — часть набора инструментов LLVM, которая поставляется вместе с утилитой LLVM AddressSanitizer, описанной в предыдущем подразделе. Чтобы использовать данный отладчик, необходимо внести небольшие изменения на этапе компиляции и заново собрать кодовую базу.

Профилировщики производительности

Иногда результаты прохождения группы нефункциональных тестов сигнализируют об ухудшении производительности. Исследование таких проблем проводится специальными инструментами. Профилировщики производительности обычно предлагают некоторые из следующих возможностей:

- сбор статистики о каждом вызове функции;
- предоставление графа вызовов функций для трассировки вызовов;
- сбор статистики относительно памяти для каждого вызова функции;
- сбор статистики о конфликтах при блокировке;
- сбор статистики о выделении/освобождении памяти;
- анализ кэша, предоставление статистики обращений к кэшу и выделение участков кода, которые плохо оптимизированы для работы с кэшем;
- сбор статистики о событиях, относящихся к многопоточности и синхронизации.

Известные наборы инструментов, которые можно использовать для профилирования производительности:

- Google Performance Tools (gperftools) — на самом деле это производительная реализация malloc, но, как утверждается на главной странице данного проекта, он предоставляет ряд средств анализа производительности, таких как heap checker.
- Callgrind (из состава Valgrind) — в основном собирает статистику о вызовах функций и отношениях между вызывающей и вызываемой функциями. Вам не нужно изменять исходный код или компоновать итоговые двоичные файлы. Инструмент используется динамически, только в сочетании с отладочной сборкой.
- Intel VTune — это пакет профилирования производительности от Intel, который поддерживает все возможности, описанные в предыдущем списке.

План лекции

Интеграция с другими языками	Модульное тестирование и отладка	Системы сборки
35 минут	35 минут	20 минут

Введение

Навыки написания программ и библиотек на языке C могут оказаться еще полезнее, чем можно было бы ожидать. Благодаря важной роли в разработке операционных систем язык C проникает в другие области. Написанные на нем библиотеки потенциально могут загружаться и применяться в других языках программирования. Пользуясь преимуществами высокоуровневых языков, вы можете задействовать в их средах эффективный и быстрый код, написанный на C.

Что такое система сборки

Система сборки — набор программ и сопутствующих текстовых файлов, которые позволяют собрать кодовую базу программного обеспечения.

Сборка кодовой базы — получение конечных продуктов компиляции из исходных файлов. Например, если говорить о C, то конечными продуктами могут быть исполняемые файлы, разделяемые объектные файлы или статические библиотеки, и задача системы сборки состоит в том, чтобы сгенерировать их из исходных файлов, из которых состоит кодовая база. То, какие именно операции нужно для этого выполнить, во многом зависит от языков программирования, на которых написан исходный код.

Еще один аспект систем сборки — их способность собирать огромные проекты с множеством модулей внутри. Конечно, то же самое можно сделать с помощью скриптов командной оболочки или путем написания рекурсивных файлов Makefile, которые могут перебирать модули на любом количестве уровней, но мы говорим о встроенной поддержке такой возможности. К сожалению, Make не умеет делать этого по умолчанию. А вот другое известное средство сборки, CMake, может нам помочь. Мы еще вернемся к этому в разделе, посвященном CMake.

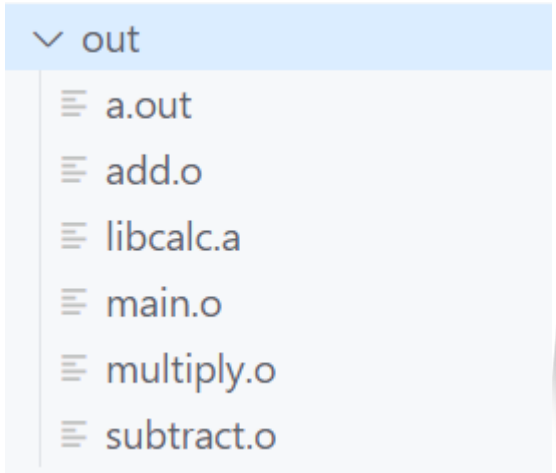
Сегодня многие проекты используют Make в качестве стандартной системы сборки, действуя через CMake. Это один из аспектов инструмента CMake, который делает его чрезвычайно важным, и вы должны научиться с ним работать, прежде чем присоединяться к проекту на C/C++. Отмечу, что CMake можно использовать не только для C и C++, но и для других языков программирования.

Make

Пример 33_2. Файлы и каталоги проекта

```
$ tree
.
├── calc
│   ├── add.c
│   ├── calc.h
│   ├── multiply.c
│   └── subtract.c
└── exec
    └── main.c
3 directories, 5 files
```

Сборка проекта




```
$ mkdir -p out
$ gcc -c calc/add.c -o out/add.o
$ gcc -c calc/multiply.c -o out/multiply.o
$ gcc -c calc/subtract.c -o out/subtract.o
$ ar rcs out/libcalc.a out/add.o out/multiply.o out/subtract.o
$ gcc -c -Icalc exec/main.c -o out/main.o
$ gcc -Lout out/main.o -lcalc -o out/a.out
```

Мы получили артефакты: статическую библиотеку libcalc.a и исполняемый файл a.out. Количество команд может расти с увеличением количества исходных файлов.

- Будем ли мы выполнять одни и те же команды на всех платформах? Некоторые детали зависят от компилятора и системной среды, поэтому команды могут варьироваться.
- Что произойдет, если в проект будет добавлен новый каталог или модуль? Придется ли менять скрипт сборки?
- Как на скрипт сборки повлияет добавление новых исходных файлов?
- Что, если нам понадобится новый продукт компиляции — библиотека или исполняемый файл?

Make

Пример 33_2. Очень простой файл Makefile для нашего проекта (Makefile-very-simple)

```
clean:
    rm -rf out
build:
    mkdir -p out
    gcc -c calc/add.c -o out/add.o
    gcc -c calc/multiply.c -o out/multiply.o
    gcc -c calc/subtract.c -o out/subtract.o
    ar rcs out/libcalc.a out/add.o out/multiply.o out/subtract.o
    gcc -c -Icalc exec/main.c -o out/main.o
    gcc -Lout -lcalc out/main.o -o out/a.out
```

Этот Makefile содержит две цели: build и clean. Каждая из них имеет набор команд, которые должны быть выполнены при ее вызове. Этот набор называют рецептом цели.
Чтобы выполнить команды, указанные в Makefile, необходимо использовать утилиту make. Вы должны сообщить ей, какую цель нужно выполнить; если этого не сделать, то по умолчанию всегда выполняется первая.

Make

Пример 33_2. Новый, но по-прежнему простой файл Makefile для проекта (Makefile-simple)

```
CC = gcc
build: prereq out/main.o out/libcalc.a
    ${CC} -Lout -lcalc out/main.o -o out/a.out
prereq:
    mkdir -p out
out/libcalc.a: out/add.o out/multiply.o out/subtract.o
    ar rcs out/libcalc.a out/add.o out/multiply.o out/subtract.o
out/main.o: exec/main.c calc/calc.h
    ${CC} -c -Icalc exec/main.c -o out/main.o
out/add.o: calc/add.c calc/calc.h
    ${CC} -c calc/add.c -o out/add.o
out/subtract.o: calc/subtract.c calc/calc.h
    ${CC} -c calc/subtract.c -o out/subtract.o
```

Мы объявили внутри Makefile переменную и использовали ее в разных местах по аналогии с CC. Переменные в сочетании с условиями позволяют писать гибкие инструкции сборки, прилагая меньше усилий по сравнению с написанием аналогичных shell-сценариев. Еще одна замечательная особенность системы Make — возможность подключать другие файлы Makefile. Каждый файл Makefile может иметь несколько целей. Цель начинается с новой строки и содержит двоеточие в конце. Все инструкции цели (рецепта) должны иметь отступы в виде символов табуляции, чтобы программа make могла их распознать.

Make

Цели обладают одним интересным свойством: могут зависеть от других целей.
Например, в приведенном выше Makefile цель build зависит от целей prereq, out/ main.o и out/libcalc.a.
Файлы Makefile более компактные и декларативные — вот почему мы используем их. Мы хотим объявить только то, что должно быть собрано, и нам не нужно знать, по какому пути пойдет процесс сборки. Нельзя сказать, что Make полностью удовлетворяет данному требованию, но с этого начинается любая полноценная система сборки.
Еще одно свойство целей в Makefile состоит в том, что они могут ссылаться на файлы и каталоги, размещенные на диске, такие как out/multiply.o, и если программа make не обнаружит в них свежих изменений с момента последней сборки, то соответствующая цель будет пропущена. Благодаря этому вы можете компилировать только те исходные файлы, которые изменились с момента последней сборки, что позволяет избежать компиляции огромного количества кода. Простое изменение заголовочного файла может инициировать компиляцию нескольких исходных файлов, которые от него зависят.
Вот почему мы пытаемся подключать в исходных файлах только необходимые заголовки и по возможности использовать предварительное объявление вместо подключения.
Предыдущий файл Makefile получился слишком многословным. Мы хотим, чтобы цели менялись при добавлении каждого нового исходного файла. Файл Makefile будет при этом меняться, но это не будет выражаться в добавлении новых целей или изменении его общей структуры, поскольку это фактически исключило бы повторное использование данного Makefile в аналогичных проектах.

Make

Пример 33_2. Новый Makefile, использующий регулярные выражения (Makefile-by-pattern)

```
BUILD_DIR = out
OBJ = ${BUILD_DIR}/calc/add.o \
            ${BUILD_DIR}/calc/subtract.o \
            ${BUILD_DIR}/calc/multiply.o
CC = gcc
HEADER_DIRS = -Icalc
LIBCALCNAME = calc
LIBCALC = ${BUILD_DIR}/lib${LIBCALCNAME}.a
EXEC = ${BUILD_DIR}/a_1.out

build: prereq ${BUILD_DIR}/exec/main.o ${LIBCALC}
    ${CC} ${BUILD_DIR}/exec/main.o -L${BUILD_DIR} -l${LIBCALCNAME} -o ${EXEC}

prereq:
    mkdir -p ${BUILD_DIR}
    mkdir -p ${BUILD_DIR}/calc
    mkdir -p ${BUILD_DIR}/exec

${LIBCALC}: ${OBJ}
    ar rcs ${LIBCALC} ${OBJ}

${BUILD_DIR}/calc/%.o: calc/%.c
    ${CC} -c ${HEADER_DIRS} $< -o $@

${BUILD_DIR}/exec/%.o: exec/%.c
    ${CC} -c ${HEADER_DIRS} $< -o $@

clean: ${BUILD_DIR}
    rm -rf ${BUILD_DIR}
```

Make

Пример 33_2. Сборка проекта с использованием заключительной версии Makefile

```
$ make
mkdir -p out
mkdir -p out/calc
mkdir -p out/exec
gcc -c -Icalc exec/main.c -o out/exec/main.o
gcc -c -Icalc calc/add.c -o out/calc/add.o
gcc -c -Icalc calc/subtract.c -o out/calc/subtract.o
gcc -c -Icalc calc/multiply.c -o out/calc/multiply.o
ar rcs out/libcalc.a out/calc/add.o out/calc/subtract.o out/calc/multiply.o
gcc out/exec/main.o -Lout -lcalc -o out/a_1.out
```

Этот файл Makefile использует регулярные выражения в своих целях. Переменная OBJ хранит список ожидаемых переносимых объектных файлов и применяется везде, где такой список может понадобиться.
Выполнение этого файла Makefile дает тот же результат, что и в предыдущих примерах, но теперь мы можем задействовать разные удобные возможности системы Make, и в конечном счете у нас получился Make-сценарий, который легко сопровождать и использовать повторно.
Вот главная ссылка на описание GNU Make — реализацию Make от проекта GNU

CMake

Пример 33_2. Иерархия проекта после добавления трех файлов CMakeLists.txt

```
$ tree
├── CMakeLists.txt
├── calc
│   ├── CMakeLists.txt
│   ├── add.c
│   ├── calc.h
│   ├── multiply.c
│   └── subtract.c
├── exec
│   ├── CMakeLists.txt
│   └── main.c
2 directories, 8 files
```

CMake — генератор скриптов сборки для других систем, таких как Make и Ninja. Написание эффективных и кросс-платформенных файлов Makefile — утомительный процесс. CMake и аналогичные инструменты вроде Autotools созданы для того, чтобы вы могли получить хорошо оптимизированные, кросс-платформенные скрипты сборки, такие как Make или Ninja.

Еще один важный аспект — управление зависимостями, которого нет в файлах Makefile. Эти генераторы умеют проверять установленные зависимости и не генерировать скрипты сборки, если какая-то из них отсутствует в системе. Проверка компиляторов и их версий, определение их местоположения и возможностей — все это выполняется перед генерацией сценария сборки.

Если системе Make нужен файл Makefile, то CMake ищет CMakeLists.txt. Наличие этого файла в проекте означает, что для генерации Makefile используется CMake. К счастью, CMake, в отличие от Make, поддерживает вложенные модули. То есть у вас может быть несколько файлов CMakeList.txt, размещенных в разных каталогах проекта, и все они будут найдены; если запустить утилиту CMake в корне проекта, она сгенерирует для каждого из них подходящий файл Makefile.

CMake

Пример 33_2. Файл CMakeLists.txt, размещенный в корневом каталоге проекта (CMakeLists.txt)

```
cmake_minimum_required(VERSION 3.8)

include_directories(calc)

add_subdirectory(calc)
add_subdirectory(exec)
```

Этот файл CMake добавляет calc в число подключенных каталогов, которые будут использоваться компилятором C во время сборки исходных файлов. Как уже говорилось ранее, он также добавляет два подкаталога, calc и exec, каждый из которых содержит собственный файл CMakeLists.txt, описывающий процедуру компиляции ее содержимого.

Пример 33_2. Файл CMakeLists.txt, размещенный в каталоге calc (calc/CMakeLists.txt)

```
add_library(calc STATIC
  add.c
  subtract.c
  multiply.c
)
```

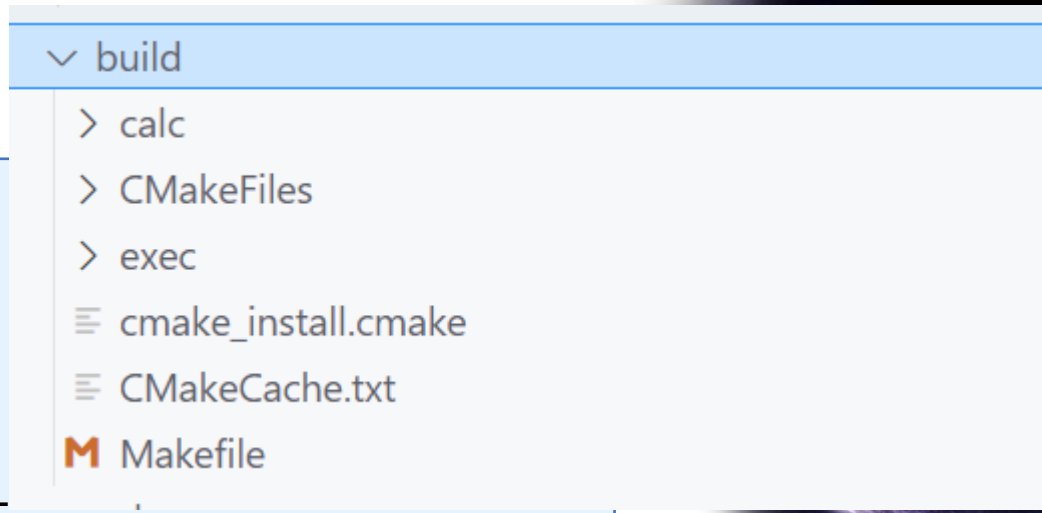
Пример 33_2. Файл CMakeLists.txt в каталоге exec (exec/CMakeLists.txt)

```
add_executable(a.out
  main.c
)

target_link_libraries(a.out
  calc
)
```

CMake

Пример 33_2. Генерация Makefile из файла CMakeLists.txt, размещенного в корневом каталоге




```
$ mkdir -p build
$ cd build
$ rm -rfv * ...
$ cmake ..
CMake Warning (dev) in CMakeLists.txt:
  No project() command is present.  The top-level CMakeL
  contain a literal, direct call to the project() command.  Add a line of
  code such as
...
-- Build files have been written to:
/mnt/c/Users/Mfili/Desktop/learningc++/imperative_programming/lections_PowerP
oint/lection_33_2/build
```

Судя по этому выводу, команда CMake сумела обнаружить рабочие компиляторы, информацию об их ABI, их возможности и т. д. В результате в каталоге build был сгенерирован файл Makefile.
Итак, у вас в каталоге build есть Makefile. Теперь можно воспользоваться командой make. Она займется компиляцией и наглядно проинформирует вас о ходе ее выполнения

CMake

Пример 33_2. Выполнение сгенерированного файла Makefile

```
$ make

[ 16%] Building C object calc/CMakeFiles/calc.dir/add.c.o
[ 33%] Building C object calc/CMakeFiles/calc.dir/subtract.c.o
[ 50%] Building C object calc/CMakeFiles/calc.dir/multiply.c.o
[ 66%] Linking C static library libcalc.a
[ 66%] Built target calc
[ 83%] Building C object exec/CMakeFiles/ex23_1.out.dir/main.c.o
[100%] Linking C executable a.out
[100%] Built target a.out
```

В настоящее время CMake используется во многих крупных проектах; вы можете собирать их исходники с помощью примерно тех же команд, которые были показаны в предыдущих терминалах. Один из таких проектов — Vim. Даже сам проект CMake собирается с помощью CMake (но только после того, как Autotools соберет минимальную систему)! У CMake есть множество версий и возможностей.

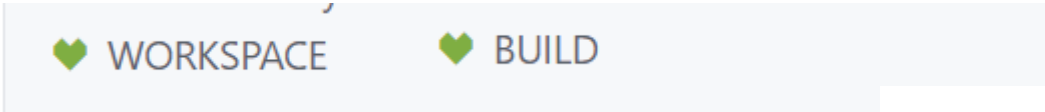
В завершение следует отметить, что CMake может создавать сценарии сборки для Microsoft Visual Studio, Xcode от Apple и других сред разработки.

Ninja

Ninja — более быстрая альтернатива системе Make. Высокая производительность достигается за счет отказа от некоторых возможностей, предлагаемых Make, таких как операции со строками, циклы и регулярные выражения.
Благодаря отсутствию этих функций Ninja имеет меньше накладных расходов. Но если вы хотите писать скрипты сборки с нуля, то данную систему лучше не использовать.
Написание скриптов Ninja можно сравнить с написанием скриптов командной оболочки. Вот почему Ninja рекомендуется использовать в сочетании со средствами генерации скриптов сборки, такими как CMake.

Пример 33_2. Выполнение сгенерированного файла Makefile

```
$ rm -rfv * ...
$ cmake -GNinja ..
CMake Warning (dev) in CMakeLists.txt:
...
-- Build files have been written to:
/mnt/c/Users/Mfili/Desktop/learningc++/imperative_programming/le
ctions_PowerPoint/lection_33_2/build
$ ninja
[6/6] Linking C executable exec/a.out
```



Bazel

Bazel — система сборки, которая была разработана компанией Google. Она изначально задумывалась быстрой и масштабируемой, способной собрать любой проект, независимо от языка программирования. Bazel поддерживает проекты на C, C++, Java, Go и Objective-C. Более того, с ее помощью можно собирать приложения для Android и iOS.

Проект Bazel стал открытым примерно в 2015 году. Это система сборки, и потому ее можно сравнивать с Make и Ninja, но не с CMake. Почти все проекты Google с открытым исходным кодом собираются с помощью Bazel, включая gRPC, Angular, Kubernetes, TensorFlow и сам Bazel.

Система Bazel написана на Java. Она славится своими параллельными и масштабируемыми сборками, что имеет большое значение в крупных проектах. Параллельная сборка также доступна в Make и Ninja; в обоих случаях для этого предусмотрен параметр -j (хотя Ninja работает параллельно по умолчанию).

Порядок использования Bazel напоминает то, как мы работали с Make и Ninja. Bazel требует наличия в проекте двух видов файлов: WORKSPACE и BUILD. Первый должен находиться в корневом каталоге, а файлы BUILD следует размещать внутри модулей, которые должны собираться в рамках одного рабочего пространства (или проекта). Это более или менее похоже на то, как работает система CMake, в которой используется три файла CMakeLists.txt, разбросанных по проекту. Однако следует помнить, что Bazel — самостоятельная система сборки, и мы не будем генерировать с ее помощью скрипты для других систем.

```
$ tree
├── WORKSPACE
├── calc
│   ├── BUILD
│   ├── add.c
│   ├── calc.h
│   ├── multiply.c
│   └── subtract.c
├── exec
│   ├── BUILD
│   └── main.c
2 directories, 8 files
```

Bazel

Пример 33_2. Файл BUILD в каталоге calc (calc/BUILD)

```
cc_library(
name = "calc",
srcs = ["add.c", "subtract.c", "multiply.c"],
hdrs = ["calc.h"],
linkstatic = True,
visibility = ["//exec:__pkg__"]
)
```

Пример 33_2. Файл BUILD в каталоге exec (exec/BUILD)

```
cc_binary(
name = "a.out",
srcs = ["main.c"],
deps = [
"//calc:calc"
],
copts = ["-Icalc"]
)
```

```
$ bazel build //...
INFO: Analyzed 2 targets (15 packages loaded, 53 targets configured).
INFO: Found 2 targets...
INFO: Elapsed time: 4.879s, Critical Path: 0.31s
INFO: 6 processes: 6 linux-sandbox.
INFO: Build completed successfully, 11 total actions
```

Bazel

```
$ tree bazel-bin
bazel-bin
├── calc
│   ├── _objs
│   │   └── calc
│   │       ├── add.pic.d
│   │       ├── add.pic.o
│   │       ├── multiply.pic.d
│   │       ├── multiply.pic.o
│   │       ├── subtract.pic.d
│   │       └── subtract.pic.o
│   ├── libcalc.a
│   └── libcalc.a-2.params
└── exec
    ├── _objs
    │   └── a.out
    │       ├── main.pic.d
    │       └── main.pic.o
    ├── a.out
    ├── a.out-2.params
    ├── a.out.runfiles
    │   ├── MANIFEST
    │   └── __main__
    │       └── exec
    │           └── a.out ->
/home/mikhail/.cache/bazel/_bazel_mikhail/115c1798e87c56b26d5ab287558c08db/execroot/__main__/bazel-
out/k8-fastbuild/bin/exec/a.out
    └── a.out.runfiles_manifest
10 directories, 15 files
```

Сравнение систем сборки

Следует понимать, что выбор системы сборки — долгосрочное решение; если вы начнете использовать в своем проекте какую-то систему, то поменять ее на другую будет непросто.

Системы сборки можно сравнивать по их различным свойствам. Управление зависимостями, способность справляться со сложными иерархиями вложенных проектов, скорость сборки, масштабируемость, интеграция с существующими сервисами, возможность добавлять новую логику и т. д. — все это можно учитывать для объективного сравнения.

Отмечу, что результаты сравнения зависят от того, для кого проводятся. Ваш выбор системы сборки должен быть основан на требованиях вашего проекта и доступных вам ресурсах.

Ссылки: 1, 2, 3.

Итоги

Было прочитано 33 лекции.
Они содержали:

- 2302 слайда
- 83 мини-темы
- 153 больших примера с кодом

Вы прорешали

- до 300 задач
- 1 проект
- 1 контрольную работу

Вы стали грамотнее!