Стандарт POSIX и программирование под UNIX

Unix as IDE, перевод

Современные системы сборки

CMake

https://cmake.org

```
cmake -G
cmake -D
cmake-gui
```

```
# CMakeLists.txt

cmake_minimum_required(VERSION 3.10)

project(Hello LANGUAGES C)

find_package(CURL REQUIRED)

include_directories(${CURL_INCLUDE_DIRS})

set(CMAKE_C_STANDARD 11)
set(CMAKE_C_STANDARD_REQUIRED True)

add_compile_options(-Wall -Wextra -Wpedantic)

add_library(my STATIC lib.c)

add_executable(hello main.c)

target_link_libraries(hello PUBLIC my ${CURL_LIBRARIES})
```

CMake tutorial

CMake cookbook

Ninja

https://ninja-build.org

> Where other build systems are high-level languages Ninja aims to be an assembler. Ninja build files are human-readable but not especially convenient to write by hand. Ninja is used to build Google Chrome, parts of Android, LLVM, and can be used in many other projects due to CMake's Ninja backend.

```
cflags = -Wall -Wextra -Wpedantic

rule cc
  command = gcc $cflags -c $in -o $out

build main.o: cc main.c
```

Meson

https://mesonbuild.com

```
project('Hello, 'c')
executable('hello, 'main.c')
```

The Absolute Beginner's Guide to Installing and Using Meson

Bazel

https://bazel.build

Bazel Tutorial: Build a C++ Project

Diff & Patch

diff(1)

patch(1)

Оригинальная версия программы была написана будущим автором языка Perl Larry Wall в 1985.

```
$ diff -u старый_файл новый_файл > разница.diff

$ patch < разница.diff

$ diff -ruN старый_каталог новый_каталог > разница.diff

$ patch -p0 < разница.diff
```

```
--- a/path/to/file  2021-01-26 22:55:55.288371691 +0300
+++ b/path/to/file  2021-01-26 22:58:31.790414616 +0300
@@ -1,5 +1,8 @@
+#include <stdio.h>
+
 int
-main(void)
+main(int argc, char** argv)
 {
+   printf("%s: Example `diff` usage;\n", __FILE__);
    return 0;
 }
```

Отладка: GDB

GDB: The GNU Project Debugger

Наиболее важные команды:

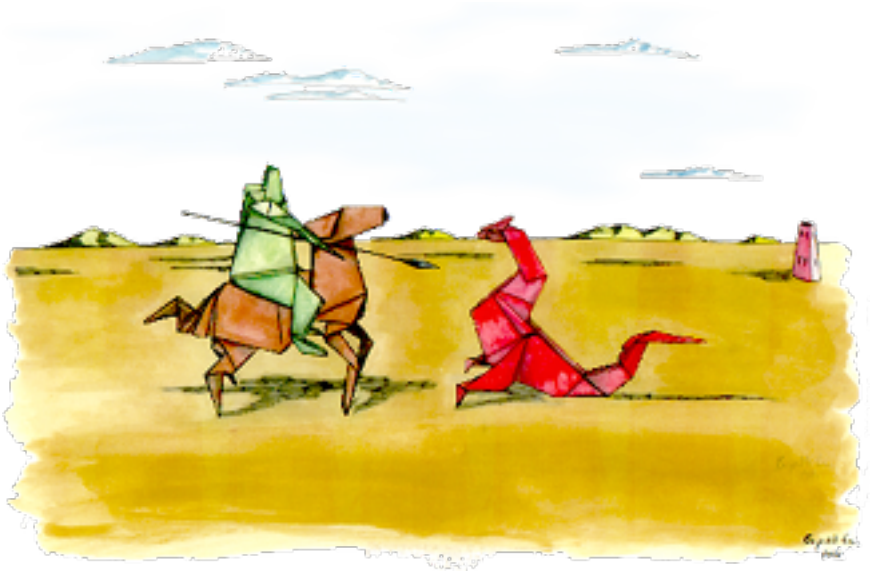
- (h)elp
- file
- (r)un
- (l)ist
- (dis)assemble
- (b)reak
- (c)ontinue, (s)tep, (n)ext
- (fin)ish
- (p)rint
- x
- backtrace (bt)
- (f)rame

Памятка по отладке при помощи GDB

GDB dashboard

Python Exploit Development Assistance for GDB

Valgrind



Что такое valgrind и зачем он нужен

Поиск ошибок работы с памятью в C/C++ при помощи Valgrind

- `memcheck`: основной модуль, обеспечивающий обнаружение утечек памяти, и прочих ошибок, связанных с неправильной работой с областями памяти — чтением или записью за пределами выделенных регионов и т.п.

- `cachegrind`: анализирует выполнение кода, собирая данные о (не)попаданиях в кэш, и точках перехода (когда процессор неправильно предсказывает ветвление). Эта статистика собирается для всей программы, отдельных функций и строк кода.
- `callgrind`: анализирует вызовы функций, используя примерно ту же методику, что и модуль cachegrind. Позволяет построить дерево вызовов функций, и соответственно, проанализировать узкие места в работе программы.
- `massif`: позволяет проанализировать выделение памяти различными частями программы.
- `helgrind`: анализирует выполняемый код на наличие различных ошибок синхронизации, при использовании многопоточного кода, использующего POSIX Threads.

```
$ valgrind --leak-check=full --show-leak-kinds=all -- ./a.out args
```

- *definitely lost*: heap-allocated memory that was never freed to which the program no longer has a pointer. Valgrind knows that you once had the pointer, but have since lost track of it. This memory is definitely orphaned.
- *indirectly lost*: heap-allocated memory that was never freed to which the only pointers to it also are lost. For example, if you orphan a linked list, the first node would be definitely lost, the subsequent nodes would be indirectly lost.
- *possibly lost*: heap-allocated memory that was never freed to which valgrind cannot be sure whether there is a pointer or not.
- *still reachable*: heap-allocated memory that was never freed to which the program still has a pointer at exit.

```
==15788== 16 bytes in 1 blocks are definitely lost in loss record 10 of 37
==15788==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==15788==    by 0x401844: get (get.c:25)
==15788==    by 0x4013DE: _get_weather_by_woeid (main.c:85)
==15788==    by 0x40169A: main (main.c:148)
==15788==
==15788== 25,662 (48 direct, 25,614 indirect) bytes in 1 blocks are definitely lost in loss record
==15788==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==15788==    by 0x4E3E947: json_object_new (json_object.c:353)
==15788==    by 0x4E3F019: json_object_new_object (json_object.c:558)
==15788==    by 0x4E431E7: json_tokener_parse_ex (json_tokener.c:349)
==15788==    by 0x4E42CF9: json_tokener_parse_verbose (json_tokener.c:200)
==15788==    by 0x4E42C9C: json_tokener_parse (json_tokener.c:188)
==15788==    by 0x4013FA: _get_weather_by_woeid (main.c:89)
==15788==    by 0x40169A: main (main.c:148)
==15788==
==15788== LEAK SUMMARY:
==15788==    definitely lost: 135 bytes in 5 blocks
==15788==    indirectly lost: 26,719 bytes in 382 blocks
==15788==      possibly lost: 0 bytes in 0 blocks
==15788==    still reachable: 0 bytes in 0 blocks
==15788==         suppressed: 0 bytes in 0 blocks
==15788==
==15788== For counts of detected and suppressed errors, rerun with: -v
==15788== Use --track-origins=yes to see where uninitialised values come from
==15788== ERROR SUMMARY: 19 errors from 15 contexts (suppressed: 0 from 0)
```

Визуализация результатов профилировки `callgrind`: KCacheGrind.

Memory error checking in C and C++: Comparing Sanitizers and Valgrind

Профилирование

Профилирование кода на C/C++ в Linux и FreeBSD

- Valgrind
- strace(1) / ltrace(1)
- bpftrace, Основы трассировки с помощью bpftrace
- GNU gprof + KCacheGrind
- Google Performance Tools
- perf
  - flame graphs
  - Linux perf Profiler UIs
  - Профилирование и трейсинг с perf

A Look at Profiling: FreeBSD Sort, перевод

Make your QEMU 10 times faster with this one weird trick