Инструменты разработчика

2019


компилятор
(g++, clang++)
Препроцессор
Компилятор
Линковщик
flex, bison
Генераторы кода
doxygen
Генераторы документации
gettext
Средства перевода на языки мира
gtest, catch2
Средства проверки корректности кода
система сборки
(meson, cmake)

Этапы сборки

Единица компиляции в C++ — файл.

```
$ ls
group.cc main.cc user.cc
$ g++ -c group.cc -o group.o          # препроцессор + компиляция
$ g++ -c user.cc -o user.o            # препроцессор + компиляция
$ g++ -c main.cc -o main.o            # препроцессор + компиляция
$ g++ main.o group.o user.o -o myprog # линковка
```

Флаги сборки

```
$ g++ -O3 ...                  # максимальная оптимизация
$ g++ -O3 -march=native ...    # оптимизация под текущую платформу
$ g++ -fsanitize=address ...   # проверка ошибок работы с памятью
$ g++ -flto ...                # оптимизация во время линковки
$ g++ -g ...                   # отладочные символы
```

Сколько всего флагов у компилятора g++?

```
$ man g++ | col -b | grep -Eo '\\s+\\-[a-zA-Z0-9\\-]+' |
sed 's/\\s*//g' | sort -u | wc -l
2580
```

Какой прирост производительности дают различные флаги?

```
using namespace std::chrono;
auto t0 = high_resolution_clock::now();
size_t n = 1<<23; // 8 млн.
std::valarray<double> x(2.0, n), y(3.0, n);
std::valarray<double> z = x + y;
auto t1 = high_resolution_clock::now();
std::cout << duration_cast<milliseconds>(t1-t0).count() << "мс\n";
```

-O0	340мс
-O3	150мс
-O3 -march=native	145мс

Как передать флаги компиляции в систему сборки?

```
$ export CXX=clang++              # во все последующие процессы
$ env CXX=clang++ meson . build  # только в следующий процесс
```

`CXX`	компилятор C++
`CXXFLAGS`	флаги компилятора C++
`CPLUS_INCLUDE_PATH`	путь к заголовочным файлам C++
`CC`	компилятор C
`CFLAGS`	флаги компилятора C
`C_INCLUDE_PATH`	путь к заголовочным файлам C
`LDFLAGS`	флаги линковщика
`LD_LIBRARY_PATH`	путь к библиотекам
`PKG_CONFIG_PATH`	путь к файлам `pkg-config`

Модульные тесты

```
int main() { return 0; } // тест пройден
int main() { return 77; } // тест пропущен
int main() { return 1; } // тест провален (любое число кроме 0 и 77)
```

mytest.cc:

```
#include <gtest/gtest.h>
TEST(my, test) { EXPECT_EQ(0,0); }
```

В терминале:

```
$ g++ mytest.cc -lgtest_main -o mytest # сборка теста
$ ./mytest                             # запуск теста
$ echo $?                              # вывести код результата
0
```

Зависимости

OpenCL.pc:

```
Name: OpenCL
Description: Open Computing Language Client Driver Loader
Version: 2.2
Libs: -L/usr/lib64 -lOpenCL
Cflags: -I/usr/include
```

В терминале:

```
$ pkg-config --cflags 'OpenCL >= 1.2'

$ pkg-config --libs 'OpenCL >= 1.2'
-lOpenCL
$ g++ $(pkg-config --cflags 'OpenCL >= 1.2') main.cc \
      $(pkg-config --libs 'OpenCL >= 1.2') -o myprog
```

meson.build:

```
OpenCL = dependency('OpenCL', version: '>=1.2')
```

Оптимизации под платформу

```
// версия 1
uint16_t byte_swap(uint16_t n) {
    return ((n & 0xff00)>>8) | ((n & 0x00ff)<<8);
}
// версия 2
uint16_t byte_swap(uint16_t n) {
    return __builtin_bswap16(n);
}
// версия 3
uint16_t byte_swap(uint16_t n) {
    #if defined(HAVE_BSWAP16)    // как определить HAVE_BSWAP16?
    return __builtin_bswap16(n);
    #else
    return ((n & 0xff00)>>8) | ((n & 0x00ff)<<8);
    #endif
}
```

Оптимизации под платформу

```
config = configuration_data() # ключ-значение
cpp = meson.get_compiler('cpp') # компилятор C++
if cpp.compiles('int main() { __builtin_bswap16(0); }')
    config.set('HAVE_BSWAP16', true)
endif
configure_file(output: 'config.hh', configuration: config)
```

config.hh:

```
#define HAVE_BSWAP16
```

Документация

```
/**
\brief
Решает систему обыкновенных дифференциальных уравнений
с правой частью \p f.
\date 2019-07-13
\param[in] f  правая часть системы уравнений
\param[in] t0 начальный момент времени
\param[in] t1 конечный момент времени
\param[in] x0 значения переменных на момент времени \f$t=t_0\f$
\return значения переменных на момент времени \f$t=t_1\f$
*/
template <class Function, class T>
Vector<T> solve(Function f, T t0, T t1, const Vector<T>& x0);
```

Документация в Meson

meson.build:

```
doxyfile = configuration_data()    # ключ-значение
doxyfile.set('OUTPUT_DIRECTORY', meson.build_root())
configure_file(input: 'Doxyfile.in', output: 'Doxyfile',
    configuration: doxyfile)    # создание файла из шаблона
doxygen = find_program('doxygen')
if doxygen.found()    # ninja doc генерирует документацию
    run_target('doc', command: [doxygen,
        join_paths(meson.build_root(), 'Doxyfile')])
endif
```

Doxyfile.in:

```
PROJECT_NAME = "My Project"
OUTPUT_DIRECTORY = @OUTPUT_DIRECTORY@
...
```

Перевод на языки мира

Версия 1:

```
std::string name;
std::cout << "Your name: ";
std::cin >> name;
std::cout << "Hello " << name << '\n';
```

Версия 2:

```
#include <libintl.h>
const char* text(const char* id) { return dgettext("myprog",id); }
int main() {
    std::string name;
    std::cout << text("Your name: ");
    std::cin >> name;
    std::cout << text("Hello ") << name << '\n';
    return 0;
}
```

В терминале:

```
xgettext --keyword=text:1 -o myprog.pot main.cc # шаблон
```

`myprog.pot` (шаблон, обновляется при каждой сборке):

```
msgid ""
msgstr ""
...
"Language: \n"
...

#: main.cc:9
msgid "Your name: "
msgstr ""

#: main.cc:11
msgid "Hello "
msgstr ""
```

myprog.ru.po (новые строки добавляются из шаблона):

```
msgid ""
msgstr ""
...
"Language: ru\n"
...

#: main.cc:9
msgid "Your name: "
msgstr "Ваше имя: "

#: main.cc:11
msgid "Hello "
msgstr "Привет, "
```

В терминале:

```
msgmerge --update myprog.ru.po myprog.pot # обновление списка строк
```

В терминале:

```
msgfmt -o myprog.ru.mo myprog.ru.po # перевод в двоичный формат
```

После установки:

```
/usr/share/locale/ru_RU/LC_MESSAGES/myprog.mo
```

Итого:

```
myprog.pot      # шаблон
myprog.ru.po    # перевод на русский язык
myprog.ru.mo    # то же самое в двоичном виде
```

Перевод на языки мира в Meson

Директория po:

```
po/LINGUAS         # список языков (ru en)
po/POTFILES        # список файлов для извлечения строк (src/main.cc)
po/myprog.pot      # шаблон
po/ru.po           # перевод на русский язык
po/en.po           # перевод на английский язык
po/meson.build     # конфигурация
```

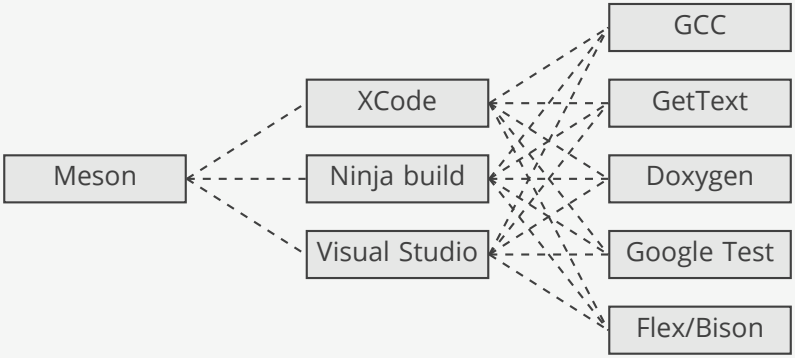
po/meson.build:

```
i18n = import('i18n')
i18n.gettext(meson.project_name(), args: ['--keyword=text:1'])
```

Команды ninja:

```
ninja myprog-pot        # создать шаблон
ninja myprog-gmo        # создать двоичные файлы
ninja myprog-update-po # обновить текстовые файлы
```

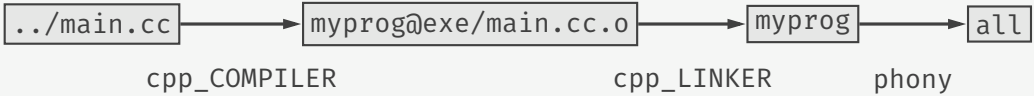
Зачем нужен Meson и Ninja?



Быстрая сборка

```
$ ninja -t deps
myprog@exe/main.cc.o: #deps 167, deps mtime 1540998419 (VALID)
    ../main.cc
    /usr/include/libintl.h
    ... # другие системные заголовочные файлы
```

Граф зависимостей:



- Компилирует только те файлы, которые зависят от изменившихся.
- Работает параллельно по графу зависимостей.
- Линкует только те библиотеки, экспортируемые символы которых изменились (с помощью Meson).

Сборка быстрая, только если программист позаботился об этом сам.

Крайность №1: один большой файл для всех классов/функций

- Нет параллелизма.
- Компилятор лучше оптимизирует.

Крайность №2: каждый класс/функция в отдельном файле

- Масса параллелизма.
- Хуже оптимизация (решается с помощью `-flto`).
- Много конкретизаций одних и тех же шаблонов.

Идеальная структура: связанные классы и функции — в отдельном файле.

Заголовочные файлы

Устаревший подход:

```
#include "vector.hh" // ищет сначала в текущей, потом в "системной"
```

Системные директории в Linux:

```
/usr/include/c++/8
/usr/include
...
```

Новый подход:

```
#include <myprog/vector.hh> // ищет в директориях, указанных в -I
```

Здесь `src` — «системная» директория:

```
g++ -Isrc src/myprog/main.cc -o build/src/myprog/main.o
```

Общая структура проекта

```
doc                     # документация
po                      # перевод на языки мира
src
├── test                # тесты
│   ├── vector_test.cc
│   ├── ...
│   └── ...
└── myproject           # основной код
    ├── vector.hh
    ├── vector.cc
    └── main.cc
```

Преимущества:

- отсутствие конфликтов,
- легко масштабируется.

Отладка

Сборка с отладочными символами:

```
g++ -g -O0 main.cc -o main.o                # ок: без оптимизации
g++ -g -O3 main.cc -o main.o                # неточные данные
g++ -g -O3 -march=native main.cc -o main.o # неточные данные
```

Запуск программы в режиме отладки:

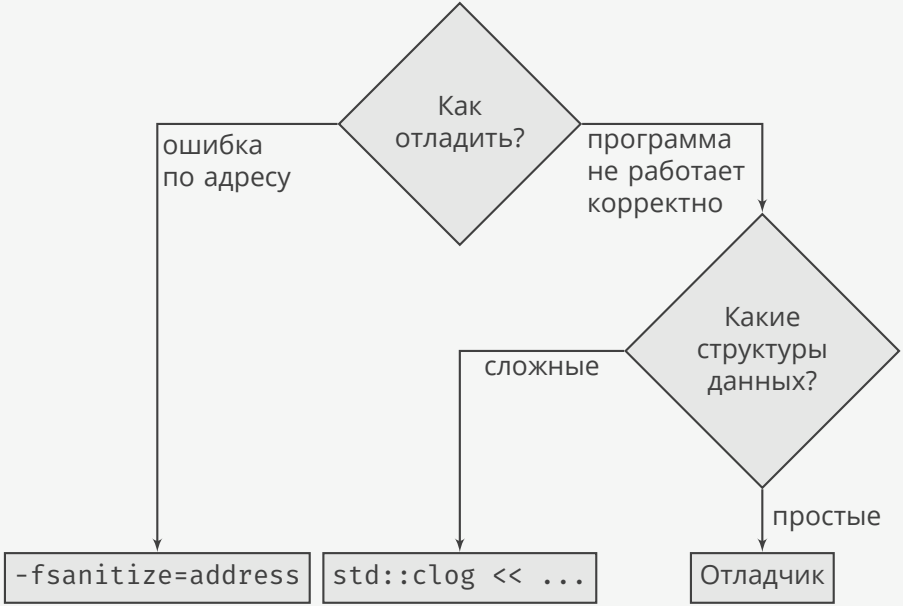
```
$ gdb ./myprog
Reading symbols from ./myprog...done.
(gdb) run
[Inferior 1 (process 3737) exited normally]
$ gdb -p НОМЕРПРОЦЕССА    # отладить уже запущенный процесс
```

Задание точки останова:

```
$ gdb ./myprog
Reading symbols from ./myprog...done.
(gdb) break main.cc:11
Breakpoint 1 at 0x400b2e: file ../main.cc, line 11.
(gdb) run
Breakpoint 1, main () at ../main.cc:11
11              std::string name;
(gdb) print name
$1 = ""
(gdb) backtrace
#0 main () at ../main.cc:11
```

Многопоточная программа:

```
$ gdb -p 26000
...
(gdb) info threads
  Id   Target Id             Frame
* 1    LWP 26140 "telegram-deskto" 0x00007f252234b559 in poll ()
...
  8    LWP 26162 "MTP::internal::" 0x00007f252234b559 in poll ()
(gdb) thread 8
[Switching to thread 8 (LWP 26162)]
#0 0x00007f252234b559 in poll () from /lib64/libc.so.6
(gdb) bt
#0 0x00007f252234b559 in poll () at /lib64/libc.so.6
#1 0x00007f2522e32b06 in g_main_context_iterate.isra
#2 0x00007f2522e32c30 in g_main_context_iteration
#3 0x000000000227a50f in ()
#4 0x0000000000000000 in ()
```


Как
отладить?
ошибка
по адресу
программа
не работает
корректно
Какие
структуры
данных?
сложные
простые
-fsanitize=address
std::clog << ...
Отладчик