

# **Руководство по написанию плагинов к компилятору GCC**



## 1.1 Введение

Начиная с версии 4.5, в компиляторе GCC появилась поддержка плагинов, поэтому теперь для добавления новой функциональности в компилятор нет необходимости его пересобирать.

Плагины представляют из себя разделяемые библиотеки (shared libraries) и в данный момент доступны только для GCC на UNIX-подобных ОС. Адекватной поддержки плагинов в MinGW нет, тем не менее их запуск в Windows возможен через WSL.

## 1.2 Установка зависимостей

В данном разделе рассказывается, как установить все зависимости на Linux Mint 18.3 с GCC 5.4.0, однако работать должно и в других версиях Mint/Ubuntu с другими версиями GCC.

- 1) Проверим, есть ли у нас в системе поддержка плагинов:

```
> gcc -print-file-name=plugin
/usr/lib/gcc/x86_64-linux-gnu/5/plugin
> ls /usr/lib/gcc/x86_64-linux-gnu/5/plugin
libcc1plugin.so ...
```

- 2) Установим зависимости для создания плагинов:

```
> sudo apt-get install gcc-5-plugin-dev
...
```

- 3) После проделанных операций в `/usr/lib/gcc/x86_64-linux-gnu/5/plugin/include` должны находиться все необходимые `.h` и `.def`-файлы.

## 1.3 Подготовка среды.

Предположим, что нам нужно написать плагин, печатающий SSA-форму программы. Назовём его `phi_debug` и создадим следующую структуру из папок и файлов:

```
.
└─ Makefile
```

```

└─ src
    └─ phi-debug
        └─ phi-debug.cpp
            └─ test
                └─ test.c

```

Простейший Makefile будет выглядеть следующим образом:

*phi-debug:*

```
mkdir -p plugin
```

```
g++ -g -Wall -Wextra -std=c++14 -I`gcc -print-file-name=plugin`/include -fPIC
-fno-rtti -shared src/phi-debug/phi-debug.cpp -o plugin/$@.so
```

*test:*

```
gcc -fplugin=plugin/phi-debug.so -O0 src/test/test.c -o src/test/test
rm src/test/test
```

*.PHONY : clean*

*clean:*

```
rm -rf plugin src/test/test
```

Таким образом:

- 1) *make phi-debug* создает папку *plugin*, а в ней будет лежать *phi-debug.so* — наш скомпиленный плагин;
- 2) *make test* запускает сборку небольшой тестовой программы с подключенным плагином;
- 3) *make clean* удаляет все бинарники;

## 1.4 Написание плагина.

Точкой входа в плагин является функция:

```
int plugin_init(struct plugin_name_args *args, struct plugin_gcc_version *version)
```

Она принимает на вход 2 аргумента:

1) Структуру с аргументами командной строки и информацией о модуле:

```
struct plugin_name_args {  
    char *base_name; // имя разделяемой библиотеки плагина без расширения .so  
    const char *full_name; // полное имя разделяемой библиотеки плагина  
    int argc; // количество аргументов командной строки.  
    struct plugin_argument *argv; // сами аргументы.  
    const char *version;           // строка версии — ее мы можем определить  
                                   // сами в plugin_init  
    const char *help;             // строка с информацией — ее мы можем  
                                   // определить сами в plugin_init  
}
```

```
struct plugin_name_args {  
    char *key; // ключ  
    char *value; // значение  
} // передача аргументов командной строки в плагин происходит через ключ  
    // -fplugin-arg-name-key1[=value1] -fplugin-arg-name-key2[=value2]
```

2) Структуру с информацией о компиляторе:

```
struct plugin_gcc_version {  
    const char *basever; // Версия GCC  
    const char *datestamp; // Дата выпуска  
    const char *devphase; // Обычно пустует  
    const char *revision; // Обычно пустует  
    const char *configuration_arguments; // Configured with: ../src/configure -v //  
    --with-pkgversion='Ubuntu 5.4.0-6ubuntu1~16.04.11' ...  
}
```

Теперь мы можем написать простейший плагин:

```

#include <gcc-plugin.h>
#include <stdio.h>

int plugin_is_GPL_compatible; // Данная переменная обязательна во всех
// плагилах, она означает, что плагин удовлетворяет GPL-лицензии (без нее
// компилятор откажется работать с плагином и упадет).

int plugin_init(struct plugin_name_args *args, struct plugin_gcc_version *version ) {
    printf("Hello, GCC!!!\n");
    return 0;
}

```

*make phi-debug* соберет плагин

*make test* выведет в одной из строк "Hello, GCC!!!"

## 1.5 Добавление функционала.

В данный момент мы умеем собирать плагин и запускать его. Мы можем работать с *plugin\_init*, как с обычной *main*-функцией, и, например, написать внутри неё простенькую игру или даже сервер)0)) Однако для возможности взаимодействия с GCC нам потребуется изучить еще несколько структур и функций:

1) Функция, регистрирующая коллбэк на GCC-событие:

```

void register_callback(const char *plugin_name, // Строка с именем плагина.
    int event, // Одно из событий GCC
    plugin_callback_func callback, // Функция-коллбэк
    void *user_data); // Некоторые пользовательский данные,
                        // передаваемые в функцию.

```

Регистрация коллбэка (вызов функции *register\_callback*) обычно происходит в функции *plugin\_init*. Сам коллбэк должен иметь следующий вид:

```

void finish_gcc(void *gcc_data, void *user_data) { ... }

```

В параметре *gcc\_data* передается некоторая служебная информация от GCC, а в *user\_data* — некоторая структура (массив, примитивный тип), задаваемая в *register\_callback* (*user\_data* может быть и *NULL*).

Событие GCC — какой-либо ключевой момент в этапе работы GCC. Например, *PLUGIN\_PASS\_EXECUTION* — запуск очередного пасса. Таким образом, наш зарегистрированный коллбэк будет вызываться перед каждым новым пасса. Больше событий можно найти вот здесь:

<https://gcc.gnu.org/onlinedocs/gccint/Plugin-API.html#Plugin-API>

## 2) Псевдо-события:

Некоторые события GCC требуют, чтобы в качестве *plugin\_callback\_func* передавался *NULL*, а в *void \*user\_data* была некоторая специальная структура. Такие события называются псевдособытиями. Всего их 3: *PLUGIN\_INFO*, *PLUGIN\_PASS\_MANAGER\_SETUP*, *PLUGIN\_REGISTER\_GGC\_ROOTS*. Мы будем работать лишь с первыми двумя.

## 3) *PLUGIN\_INFO*

Псевдособытие *PLUGIN\_INFO* требует, чтобы в качестве коллбэка передавался *NULL*, а в качестве пользовательских данных структура:

```
struct plugin_info {  
    const char *version; // Строка с версией плагина.  
    const char *help;    // Строка с информацией о плагине.  
};
```

Пример:

```
#define PLUGIN_VERSION "1.0.0"  
#define PLUGIN_HELP  "this plugin shows:\n"  
    "* basic blocks;\n"  
    "* GIMPLE instructions;\n"  
    " - arithmetic operations;\n"
```

```

" - phi-functions;\n"
" - branches;\n"
" - memory operations;"

```

```

static struct plugin_info phi_debug_plugin_info = {
    .version = PLUGIN_VERSION,
    .help =    PLUGIN_HELP,
};
// ...
// где-то в plugin_init:
register_callback(args->base_name, PLUGIN_INFO, NULL, &my_plugin_info);

```

Теперь согласно документации GCC, при запуске компилятор с этим модулем и флагом `—help / —version`, будет выведена информация о плагине, однако на версии 5.4.0 это не работает.

#### 4) PLUGIN\_PASS\_MANAGER\_SETUP

Псевдособытие *PLUGIN\_PASS\_MANAGER\_SETUP* координирует работу нашего плагина и позволяет назначить четкое место выполнения (после (перед, вместо) какого пасса запускать наш плагин, что ему требуется для работы и тд).

Для этого он принимает в качестве пользовательского параметра структуру:

```

struct register_pass_info {
    opt_pass *pass; // указатель на реализацию pass`а.
    const char *reference_pass_name; // имя пасса, к которому мы будем
    // «цепляться». Например, для прикрепления к пассу tree-ssa-dce будет имя
    // "dce", для прикрепления к пассу перегонки в ssa - "ssa".
    int ref_pass_instance_number; // сколько раз запускать плагин, при встрече
    // некоторого пасса, к которому мы прикрепляемся (0 — всегда, 1 — один //
    раз, 2 — два раза и тд)

```

```
enum pass_positioning_ops pos_op; // Позиция плагина (запуска после, до,
вместо reference_pass_name).
};
```

```
enum pass_positioning_ops {
    PASS_POS_INSERT_AFTER, // После
    PASS_POS_INSERT_BEFORE, // До
    PASS_POS_REPLACE      // Вместо
};
```

Бездумное использование *PASS\_POS\_REPLACE* может не дать GCC скомпилировать программу. Например, заменить пасс «dce» на свой собственный дебаг-пасс вполне корректно, потому что «dce» лишь выполняет оптимизации, но вот заменять пасс «ssa» нельзя, так как SSA-форма не построится и дальнейшая сборка программы будет невозможна.

*opt\_pass \*pass*: напомним, что в данный момент GCC активно переписывается с C на C++, поэтому в него завезли наследование, виртуальные методы и тд. В *opt\_pass \*pass* нужно передать свой собственный класс, унаследованный от класса *gimple\_opt\_pass*, с определенным конструктором и 2-мя переопределенными виртуальными методами:

```
struct phi_debug_pass : gimple_opt_pass {
    phi_debug_pass(gcc::context*ctx) : gimple_opt_pass(phi_debug_pass_data, ctx)
}
virtual unsigned int execute(function*fun) override;
virtual phi_debug_pass *clone() override;
};
```

Метод *execute* запускается при заходе в очередную функцию. Метод *clone* используется для копирования пасса и нами использован не будет, поэтому реализуем их вот так:



```

unsigned int phi_debug_pass::execute(function*fn) { /* Do something great */ return
0; }
phi_debug_pass *phi_debug_pass::clone() { return this; }

```

В конструктор базового класса должны передаваться контекст GCC и структура `pass_data`. Контекст — это просто глобальная переменная `g`, а `pass_data` имеет следующий вид:

```

struct pass_data {
    enum opt_pass_type type; // Тип оптимизации.
    const char *name; // Имя пасса.
    unsigned int optinfo_flags; // Флаги оптимизации для ключа GCC -fopt-info
    timevar_id_t tv_id; // Здесь и далее различные флаги, которые мы просто
                        // занулим за ненадобностью или заполним дефолтами.
    unsigned int properties_required;
    unsigned int properties_provided;
    unsigned int properties_destroyed;
    unsigned int todo_flags_start;
    unsigned int todo_flags_finish;
};

enum opt_pass_type {
    GIMPLE_PASS, // пассы, так или иначе работающие с SSA (то что нам
                // надо)
    RTL_PASS, // RTL работает после SSA пассов
    SIMPLE_IPA_PASS, // не нужно
    IPA_PASS // не нужно
};

```

## 1.6 phi-debug

Теперь мы можем написать плагин, который делает что-то полезное, например, печатает названия всех встреченных на своем пути функций, и запускающийся после перегонки кода в SSA:

```
// Copyright (C) 2019 Mikhail Masyagin
```

```
// Перечисленных хэдеров достаточно для выполнения 1 лабы.
```

```
#include <iostream>
```

```
#include <sstream>
```

```
#include <string>
```

```
#include <gcc-plugin.h>
```

```
#include <plugin-version.h>
```

```
#include <coretypes.h>
```

```
#include <tree-pass.h>
```

```
#include <context.h>
```

```
#include <basic-block.h>
```

```
#include <tree.h>
```

```
#include <tree-ssa-alias.h>
```

```
#include <gimple-expr.h>
```

```
#include <gimple.h>
```

```
#include <gimple-ssa.h>
```

```
#include <tree-phinodes.h>
```

```
#include <tree-ssa-operands.h>
```

```
#include <ssa-iterators.h>
```

```
#include <gimple-iterator.h>
```

```
#define PREFIX_UNUSED(variable) ((void)variable) // Макрос, чтобы
```

*// компилятор не ругался на неиспользуемые переменные.*

*int plugin\_is\_GPL\_compatible = 1;*

*#define PLUGIN\_NAME "phi-debug" // имя*

*#define PLUGIN\_VERSION "1.0.0" // версия*

*// информация о плагине.*

*#define PLUGIN\_HELP "this plugin shows:\n"*

*"\* basic blocks;\n"*

*"\* GIMPLE instructions;\n"*

*" - arithmetic operations;\n"*

*" - phi-functions;\n"*

*" - branches;\n"*

*" - memory operations;"*

*static struct plugin\_info phi\_debug\_plugin\_info = {*

*.version = PLUGIN\_VERSION,*

*.help = PLUGIN\_HELP,*

*};*

*static const struct pass\_data phi\_debug\_pass\_data = {*

*.type = GIMPLE\_PASS,*

*.name = PLUGIN\_NAME,*

*.optinfo\_flags = OPTGROUP\_NONE, // Мы ничего не оптимизируем*

*.tv\_id = TV\_NONE,*

*.properties\_required = PROP\_gimple\_any,*

*.properties\_provided = 0,*

*.properties\_destroyed = 0,*

```
.todo_flags_start = 0,  
.todo_flags_finish = 0,  
};
```

```
struct phi_debug_pass : gimple_opt_pass {  
    phi_debug_pass(gcc::context*ctx) : gimple_opt_pass(phi_debug_pass_data, ctx)  
    {}  
    virtual unsigned int execute(function*fun) override;  
    virtual phi_debug_pass *clone() override { return this; }  
};
```

```
static unsigned int phi_debug_function(function*fn)  
{  
    std::cout << "func: " << "\"" << function_name(fn) << "\"" << " {" <<  
std::endl;  
    std::cout << "}" << std::endl;  
    std::cout << std::endl;  
    return 0;  
}
```

```
unsigned int phi_debug_pass::execute(function*fn) { return  
phi_debug_function(fn); }
```

```
static struct register_pass_info phi_debug_pass_info = {  
    .pass = new phi_debug_pass(g), // тот самый глобальный  
контекст  
    .reference_pass_name = "ssa", // Прикрепимся к ssa-наcssу  
    .ref_pass_instance_number = 1, // 1 прогонка  
    .pos_op = PASS_POS_INSERT_AFTER, // После ssa-насса.  
};
```

```

int plugin_init(struct plugin_name_args *args, struct plugin_gcc_version *version)
{
    if(!plugin_default_version_check(version, &gcc_version)) { // Проверка версии
        return 1;
    }

    register_callback(args->base_name, PLUGIN_INFO, NULL,
&phi_debug_plugin_info);
    register_callback(args->base_name, PLUGIN_PASS_MANAGER_SETUP, NULL,
&phi_debug_pass_info);

    return 0;
}

```

## 1.6 несколько советов

- 1) Возможно так только у меня, но «dce», а следовательно и плагин, привязанный к нему, работает у меня лишь при -O3 оптимизации. В то же время закрепление после «ssa» работает всегда, даже при -O0;
- 2) Как ни странно, но да, плагин для GCC, компилятора C, мы будем писать на C++14;
- 3) Чтобы было легче оценить, насколько хорошо работает ssa-debug, можно скомпилировать тестовую программу test.c с помощью следующей команды:

```
gcc -O0 -fdump-tree-ssa-graph -g test.c -o test
```

В таком случае помимо бинарника будут созданы 2 файла: *test.c.018t.ssa* и *test.c.018t.ssa.dot*. Первый текстовый, а второй можно посмотреть в виде графа через *xdot*. Примерно к их результатам работы и надо стремиться;
- 4) Насколько я могу судить, в 7 версии GCC API работы с деревом (tree) немного поменялся, но там все легко поправить;

- 5) Чтобы не мучиться с поиском всяких макросов и функций, заходите в */usr/lib/gcc/x86\_64-linux-gnu/5/plugin* и делаете *grep -nr «то что я хочу найти»*.