GCC internal programming tips

Wen ZHANG

August 6, 2010

Contents

1 Licence

2 Introduction

The information provided in the official GCC Internal manual is far from necessary for developers to add their own features to GCC, so I conclude all my experience in hacking gcc to this article. You can see this as a complementary part to the internal manual. Hope this article can provide some useful tips to you.

By the way, if not write explicitly, the functions and methods are used in GIMPLE stage.

3 Declare types

In this section, I will show some tips about how to declare types manually by the developers in GCC.

3.1 Declare pointer types

```
build_pointer_type(tree type)
```

This is used to declare the pointer type of type `type`
For example:

```
ptr_integer_type_node = build_pointer_type(integer_type_node);
```

3.2 Declare qualified types

```
build_qualified_type(tree type, int type_quals)
```

This is used to declare a qualified type of type `type`, the qualifiers are decided by type_quals, which are

1. `TYPE_UNQUALIFIED`
2. `TYPE_QUAL_CONST`
3. `TYPE_QUAL_VOLATILE`
4. `TYPE_QUAL_RESTRICT`

We can use `|` to declare multiple qualifiers at the same time.

For example:

```
const_integer_type_node = build_qualified_type(integer_type_node , TYPE_QUAL_CONST)
```

3.3 Declare mixed types

We can combine the above method to declare a mixed type. For example, to declare a `const int *` type, we need the following statement:

```
const_ptr_pointer_type_node = build_qualified_type(build_pointer_type(integer_type_node)
    , TYPE_QUAL_CONST);
```

3.4 Declare structures

To declare a structure, we need to follow two steps:

1. Use build_decl() (see section 4.1 for details) to generate fields, and use `TREE_CHAIN()` to chain all the fields together. For example, we need to declare a structure like figure 1, we need to use method in figure 2 to generate all the fields.

```
struct example_structure
{
    int field1;
    char field2;
};
```

Figure 1: example sturcture

```
tree field, fields;

fields = build_decl(BUILTINS_LOCATION, FIELD_DECL, get_identifier("field1"),
    integer_type_node);

field = build_decl(BUILTINS_LOCATION, FIELD_DECL, get_identifier("field2"),
    char_type_node);
TREE_CHAIN(field) = fields;
fields = field;
```

Figure 2: Build example structure fields

2. Declare a record type, associate the fields with the type, and layout the new structure type. Luckily, the latter two steps are encapsulated in the function `finish_builtin_struct()`. To finish declaring the structure in figure 1, we need the codes in 3 to handle the left works.

```
tree struct_type;

struct_type = lang_hooks.types.make_type(RECORD_TYPE);
finish_builtin_struct(struct_type, "example_structure", fields, NULL_TREE);
```

Figure 3: Finish declaring the structure

4 Declare variables

In this section, I will introduce some practical tips to declare local and global variables in GCC.

4.1 Declare local variables

There are several ways to declare a variable, but at first I want to introduce a very low level method, and then to the high-level functions.

1. `tree build_decl(location_t loc, enum tree_code code, tree name, tree type)` We can use this function to declare a variable in type `type` with name `name`, and with all its properties, such as initial value, whether static or not, whether external or not, in their defaults, maybe 0. These 0s of course have different meanings for different properties. You can check `tree.h` for detailed explanations. This function has four parameters, `loc` is the location information for debugging I guess. `code` refers to the kind of the node, and usually is `VAR_DECL` if we want to declare a variable node. `name` is the name of the variable, this is not a string, and we'd better use `get_identifier()` function to generate it. `type` for a variable declaration is the type of the variable, for example if we want to declare a integer, here can be `integer_type_node`. The return is the declared variable. The next step is modifying its properties following our purpose.

2. `tree create_tmp_var_raw(tree type, const char *prefix)` This function is a higher level wrapper of `build_decl()`. It can be used to declare a local variable in a function by modifying the attributes of the node. `type` is the type of the variable. `prefix` is the prefix of the variable name which is in the form of `prefix` + `.` + `index no.`. The return is the declared variable.

3. `tree make_rename_temp(tree type, const char *prefix)` This function is higher than and indirectly wraps `create_tmp_var_raw()`. This function does more work to add the newly declared variable, and requires we are in GIMPLE stages I guess. So there's no need to use this function in earlier stages, or it may cause some problem. The parameters and the return are the same as that of the function `create_tmp_var_raw()`.

4.2 Declare global variables

To declear a global variable, we should notice two points.

1. global variables should be defined in the file scope, and in order to add a variable to the file scope, a good place I know is around invoking `PLUGIN_START_UNIT` plugin callback; or add it in the function handling this plugin event if using the plugin. There may be some methods can achieve the same effect in the function level, but I am not sure.
2. we can use the `build_decl()` function to declare the variable, and then set its properties. If the variable is a extern one, `TREE_PUBLIC` and `DECL_EXTERNAL` should be 1, and `TREE_STATIC` should be 0; otherwise, `TREE_PUBLIC` and `TREE_STATIC` should be 1, and `DECL_EXTERNAL` should be 0.

```
tree fn_type_list;

fn_type_list = build_function_type_list(integer_type_node , integer_type_node ,
    build_pointer_type(struct_usage), NULL_TREE);
```

Figure 4: Build return and parameter type list for the function

```
tree fn;

fn = build_fn_decl("getrusage", fn_type_list);
```

Figure 5: Declare function

5 Declare functions

In this section, I will elaborate the way to declare a function. Note that this method is used to declare a function only, but will not instrument function. We can use this method only when the function is existed in the program or can be called from an external library linked with the program.

To declare a function, we have two steps:

1. Declare the return variable type, and all parameter types. We can use `build_function_type_list()` to achieve this purpose, the parameters are the first one is the return type of the function to declare, and the left are the parameters of the function to declare from left to right. The end of parameter list is indicated by a `NULL_TREE`. For example, we need to declare the function `int getrusage(int who, struct rusage *usage)`, suppose we have already declared the type `struct rusage *usage` as `struct_rusage`, the code to declare the return and parameter list is as figure 4.
2. Use `build_fn_decl()` function to declare the function. The first parameter is the function name; the second parameter is the return and parameter type we generated in the last step. See figure 5.

```
#include "hashtab.h"

htab_t pointer_htab;

pointer_htab = htab_create(1024, htab_hash_pointer, htab_eq_pointer, NULL);
```

Figure 6: Create a simple pointer hash table

6 Built-in data structures

GCC source code provides several implemented advanced data structures. I want to introduce two of them, hash table and splay tree. These two data structures are in libiberty.

6.1 Hash table

A brief introduction of hash table can be found in `http://en.wikipedia.org/wiki/Hash_table`. I will only discuss several hash table interfaces provided by the libiberty. The header file is at the "include/hashtab.h". The source file is at the "libiberty/hashtab.c". This hash table can store `<key, value>` pairs, and `key` is the hash value of something, for example the hash value of the `value`.

6.1.1 Create a hash table

We can use the function `htab_t htab_create_alloc(size_t, htab_hash, htab_eq, htab_del, htab_alloc, htab_free)` to allocate a hash table. The first argument is the initial number of the slots in the hash table. But the total amount of the slots can vary dynamically while using the hash table. The second argument is a function hash table used to do the hashing and generate the key. This is a user defined function in the form of `hashval_t functionName(const void *)`. The third argument is a function hash table used to do the comparison between keys. This is a user defined function in the form of `int functionName(const void *, const void *)`. The fourth argument is a function will be called when user requiring removing some entry from the hash table. This is a user defined function in the form of `void functionName(void *)`. This function is not necessarily needed, and can be set to `NULL` sometimes. The fifth and the sisth arguments are the user defined functions for allocation and free of the hash table. The fifth argument is similar to `calloc`, and in the form of `void *functionName(size_t, size_t)`. The sixth argument is in the form of `void functionName(void *)`.

There is a simplified version of `htab_create_alloc()`, which is `htab_t htab_create(size_t, htab_hash, htab_eq, htab_del)`. The `htab_alloc` and the `htab_free` are using gcc internal allocation and free mechanism.

This hash table structure has already implementeda simple `htab_hash` function , and a `htab_eq` function for pointers, which are `htab_hash_pointer` and `htab_eq_pointer`. So we can directly use them when we need to use hash table to handle pointers. Further more, there is a `htab_hash_string` function which can be used as `htab_hash` for strings. Here is a very simple case for the user to create a hash table using pointers as keys, see figure 6.

6.1.2 Find, Insert and Delete items from the hash table

Both Insert and Delete operations are all depending on the Find operation for the hash table. In this implementation, find function has two modes, one is find, the the other is find and insert. So if we want to find or want to delete some item, we need to use find mode, and if we want to do the insert operation, we need to use the find and insert mode.

There are four functions for find.

1. `PTR htab_find_with_hash(htab_t htab, const PTR element, hashval_t hash)`
2. `PTR htab_find(htab_t htab, const PTR element)`
3. `PTR * htab_find_slot_with_hash(htab_t htab, const PTR element, hashval_t hash, enum insert_operation insert)`
4. `PTR * htab_find_slot(htab_t htab, const PTR element, enum insert_operation insert)`

Note that `PTR` is an alias of `void *` for ansi C definition and `char *` for traditional C definition. The definition for `enum insert_operation` is `enum insert_operation` {`NO_INSERT, INSERT`}.

The first two function is a group of function which only return the found item, however, the latter two return the address of the found item. So for insert and delete operations, we need to use the latter two to locate the slot. for the first and the third functions, the second argument is the `value` of the pair, and the third argument hash is the `key` of the pair. The second and the fourth functions are the simplified versions respectively. They use the hash value of the `value` from the pair as the `key`.

In order to insert a pair into the hash table, we need to use either of the last two functions, and set the `insert INSERT`. The function will automatically insert the pair into the proper place in the hash table, or even expanded it if there is not enough places.

To delete a pair, we need to use either of the last two functions, and set the `insert NO_INSERT`, to locate the pair. Then we need to use `void htab_clear_slot(htab_t htab, PTR *slot)` to clear the contents in the slot. For simplicity, here are two functions encapsulate the two steps of deleting a pair.

1. `void htab_remove_elt(htab_t htab, PTR element)`
2. `void htab_remove_elt_with_hash(htab_t htab, PTR element, hashval_t hash)`

The difference of the two is the former one use the hash of the `element`(`value`) as the `hash`. The latter one requires user inputing a hash value(`key`) manualy.

6.1.3 Traverse a hash table

Sometimes we need to traverse the hash table and apply some actions to each pair. Libiberty's implementation provides this traverse function, we can directly use it. There are two functions for traversing.

1. `void htab_traverse_noresize(htab_t htab, htab_trav callback, PTR info)`
2. `void htab_traverse(htab_t htab, htab_trav callback, PTR info)`

Both functions will traverse the hash table `htab`. For each found pair, it will apply the function `callback`, which is `int functionName(void **, void *)`, and it will pass `info` to the callback function as its second argument. This helps pass user-defined data into the callback function. If the callback function returns 0, then the traverse will be aborted, otherwise it continues traversing.

6.1.4 Dispose a hash table

We can call `void htab_delete(htab_t htab)` to dispose a hash table.

6.1.5 Miscellaneous

Here are parts of other functions.

1. `size_t htab_size(htab_t htab)`. Return the current size of the hash table.
2. `size_t htab_elements(htab_t htab)`. Return current number of elements(pairs) in the hash table.

6.2 Splay tree

Splay tree is a balanced tree structure, and a brief introduction of it can be found in `http://en.wikipedia.org/wiki/Splay_tree`. I will introduce the splay tree interfaces provided by the libiberty. The header file is at "include/splay-tree.h" and the source file is at "libiberty/splay-tree.c". Note that each node of the tree is a `<key, value>` pair.

6.2.1 Create and delete a splay tree

There are two functions for creating a splay tree:

1. `splay_tree splay_tree_new(splay_tree_compare_fn, splay_tree_delete_key_fn, splay_tree_dele`
2. `splay_tree splay_tree_new_with_allocator(splay_tree_compare_fn, splay_tree_delete_key_fn, splay_tree_delete_value_fn, splay_tree_allocate_fn, splay_tree_deallocate_fn, void *)`

Both functions require user provide a compare function `splay_tree_compare_fn` for comparing the keys. `splay_tree_delete_key_fn` and `splay_tree_delete_value_fn` are optional functions provided by the user. They will be called when removing a node from the tree. The second function enable user to provide additional `splay_tree_allocate_fn` and `splay_tree_deallocate_fn` functions to override the default allocation and deallocation scheme. The last parameter is for passing additional user data to the allocation and deallocation functions.

The libiberty implementation has already provides two comparing functions for integers and pointers. They are `int splay_tree_compare_ints(splay_tree_key, splay_tree_key)` and `int splay_tree_compare_pointers(splay_tree_key, splay_tree_key)`. Either of them can serve as `splay_tree_compare_fn`

The function `void splay_tree_delete(splay_tree)` is used for delete a splay tree.

6.2.2 Insert, remove and lookup elements in the splay tree

To insert a `<key, value>` pair into the splay tree, we can use the function `splay_tree_node splay_tree_insert(splay_tree, splay_tree_key, splay_tree_value)`.

We can use the function `void splay_tree_remove(splay_tree, splay_tree_key)` to remove a pair of data from the splay tree.

We can also use the function `splay_tree_node splay_tree_lookup(splay_tree, splay_tree_key)` to find a pair in the splay tree.

6.2.3 Miscellaneous

Here are some other functions for the splay tree.

1. `splay_tree_node splay_tree_predecessor(splay_tree, splay_tree_key)`
2. `splay_tree_node splay_tree_successor(splay_tree, splay_tree_key)`
3. `splay_tree_node splay_tree_max(splay_tree)`
4. `splay_tree_node splay_tree_min(splay_tree)`
5. `int splay_tree_foreach(splay_tree, splay_tree_foreach_fn, void *)`

I have not tried them yet, so you can try them as needed.

Constant	Explanation
char_type_node	char type
signed_char_type_node	signed char type
unsigned_char_type_node	unsigned char type
short_integer_type_node	short integer type
short_unsigned_type_node	short unsigned integer type
integer_type_node	integer type
unsigned_type_node	unsigned integer type
long_integer_type_node	long integer type
long_unsigned_type_node	long unsigned integer type
long_long_integer_type_node	long long integer type
long_long_unsigned_type_node	long long unsigned integer type
uint32_type_node	uint32_t type
uint64_type_node	uint64_t type
float_type_node	float type
double_type_node	double type
long_double_type_node	long double type
float_ptr_type_node	float * type
double_ptr_type_node	double * type
long_double_ptr_type_node	long double * type
integer_ptr_type_node	int * type
void_type_node	void type
ptr_type_node	void * type
const_ptr_type_node	const void * type
size_type_node	size_t type
pid_type_node	pid_t type
fileptr_type_node	FILE * type

Table 1: Constant type nodes

7 Constants

In this section, I will introduce some constants maybe used while programming. The constant nodes for types are in table 1, and for values are in table 2. These constant nodes can be found in "gcc/tree.h". There are also many builtin functions that can be directly used, such `printf()`. These function indices can be found in "gcc/builtins.def". To get the declaration of the functions, user need to use the `BUILT_IN_XXX` for each function as index, and locate them in the built_in_decls[ ] array. For example, if we want to generate GIMPLE statment to use `printf()` to print out the "Hello, world!", we need to following the code in figure 7.

Constant	Explanation
integer_zero_node	(int)0
integer_one_node	(int)1
integer_minus_one_node	(int)-1
size_zero_node	(size_t)0
size_one_node	(size_t)1
null_pointer_node	NULL

Table 2: Constant value nodes

```
tree hello_world_str;
/* suppose "Hello, world!" string has already stored in the hello_world_str */
tree fn_printf;
gimple stmt;

fn_printf = built_in_decls[BUILT_IN_PRINTF];
stmt = gimple_build_call(fn_printf, 1, hello_world_str);
```

Figure 7: Use builtin `printf()` to generate a GIMPLE statement which can print out "Hello, world!"

8 Maintain CFG

This section will introduce the steps of maintaining the CFG.

8.1 Create basic blocks

In order to modify the control flow graph, we need to create basic blocks sometimes. There are at least two kinds of "creating", one is spliting the existing basic block, and the other is creating a basic block from zero. Spliting a basic block is easy, we can simply use the function `edge split_block(basic_block bb, void *i)`. The explanation for the function is "Splits basic block BB after the specified instruction I (but at least after the labels). If I is NULL, splits just after labels. The newly created edge is returned. The new basic block is created just after the old one.", from "gcc/cfghooks.c". This function will take care maintaining the CFG, so just feel ease to use it. Creating a basic block from zero is also easy, but we need to maintain the CFG manually. Normally, we can use the function `basic_block create_basic_block(void *head, void *end, basic_block after)` to create a basic block. Usually the first argument is the a gimple sequence(gimple_seq) containing the statements for the block, the second argument is NULL, and the third is a basic block that the newly create basic block will be linked after(all basic blocks are stored in a link list, which is not necessarily having relationships with the CFG). There are some attributes of the basic block I'm not sure whether need to be set for the newly created ones. Here are they:

1. `discriminator`. I usually set it one larger than the previous block.

For each statement in the newly created block `A`, user need to use the function `void gimple_set_bb(gimple stmt, basic_block bb)` to set it belonging to the `A`.

8.2 Maintain control flow graph

After creating a basic block, we need to insert it into the control flow graph. You can learn the detail from the "cfg" pass, which is from "gcc/tree-cfg.c", of cource. Here I will notice some important and useful tips. One of the key points of maintaining the control flow grpah is maintaining the edges between the basic blocks. Here are some functions for creating, removing, and redirecting edges.

1. `edge make_edge(basic_block src, basic_block dest, int flags)`. Make a edge from `src` to `dest`. The edge type is notified by `flags`, which can be `EDGE_FALLTHRU` if `dest` is the direct son of the `src` in CFG, `EDGE_TURE_VALUE` if `src` is end of a branch statement, for example `if...cond`, and `dest` is the block to execute when the branch condition is ture, `EDGE_FALSE_VALUE` is similar, and so on. The rest possible flag values can be found at the file "gcc/basic-block.h".
2. `edge find_edge(basic_block pred, basic_block succ)`. Find the edge from `pred` to `succ`. If there's no such edge, return NULL. This function can be found at the file "gcc/cfganal.c".
3. `void remove_edge(edge e)`. Remove the edge `e` from the CFG.
4. `edge redirect_edge_and_branch(edge e, basic_block dest)`. From "gcc/cfghooks.c", "Redirect edge `e` to the given basic block `dest` and update underlying program representation. Returns edge representing redirected branch (that may not be equivalent to E in the case of duplicate edges being removed) or NULL if edge is not easily redirectable for whatever reason.".

For edges representing "goto", for example out edges of if...cond statement, user needs to set the edge's member variable `goto_locus` and `goto_block`. Developer can use the marco `FOR_EACH_EDGE()` to traverse the edge of a given basic block, you can check the official "GCC Internal Manual" for detail. There are also a bunch of other helpful functions that I have not listed. So you can check "gcc/cfghooks.c" for more information.

8.3 Maintain dominance tree

Other than maintaining the CFG, we should maintain the dominance information between basic blocks as well. There are two kinds of dominance directions, dominance(`CDI_DOMINATORS`) and post dominance(`CDI_POST_DOMINATORS`). Usually we do not care post dominance. So in following parts, dominance direction refers to the dominance(`CDI_DOMINATORS`) if not specified explicitly. If one block A is the direct son of another block B in CFG, we say B is dominating A, or A is dominated by B, or B is the dominator of A. In GCC, this dominance information is generated by using DFS(depth first search) travelling the CFG. There should not be any loops in dominance information, and one blocks should not be dominated by more than one blocks. So one method is clearing all the dominance informationa after modifying the CFG, and ask GCC to rebuild it, but this method is slow and inefficient. Another one is maintaining the dominance information manual when modifying the CFG.

Here are some very useful functions for maintaining the dominance information manually:

1. `basic_block get_immediate_dominator(enum cdi_direction dir, basic_block bb)`. Giving a dominance direction(`CDI_DOMINATORS`, `CDI_POST_DOMINATORS`), and a basic block, return the dominator of bb according to the direction.

```
int i;
for(i = 0; i < 10; i ++);
```

Figure 8: A loop example which will be inserted into a program

2. `void set_immediate_dominator(enum cdi_direction dir, basic_block bb, basic_block dominated_by)`. Set the dominator of `bb` to `dominated_by` according to the `dir`.
3. `void redirect_immediate_dominators(enum cdi_direction dir, basic_block bb, basic_block to)`. Set blocks' dominators which are `bb` to `tt`.

For more detail, please check the file "gcc/dominance.c".

8.4 Example

This subsection will give an example of inserting the code in the figure 8 into a program, suppose between `start_bb` and `end_bb`, and suppose the edge from `start_bb` to `end_bb` is the fall-through edge. The code of the instrumentation is in the figure 9.

```
basic_block bb, new_bb, cond_bb, then_bb, else_bb, first_new_bb;
tree counter_i;
gimple_stmt_iterator gsi;
gimple_seq gseq;
gimple gstmt;
edge e;

counter_i = make_rename_temp(integer_type_node, "i");
bb =  start_bb;
else_bb = end\_bb;

/* remove edge from start_bb to end_bb */
e = find_edge(start_bb, end_bb);
if(e != NULL)
    remove_edge(e);

/* insert i = 0 */
gseq = gimple_seq_alloc();
gstmt = gimple_build_assign(counter_i, integer_zero_node);
gimple_seq_add_stmt(&gseq, gstmt);
new_bb = create_basic_block(gseq, NULL, bb);
new_bb->discriminator = bb->discriminator + 1;
e = make_edge(bb, new_bb, EDGE_FALLTHRU);
bb = new_bb;
first_new_bb = new_bb;
for(gsi = gsi_start_bb(bb); !gsi_end_p(gsi); gsi_next(&gsi))
    gimple_set_bb(gsi_stmt(gsi), bb);

/* insert i = i + 1 */
gseq = gimple_seq_alloc();
gstmt = gimple_build_assign_with_ops(PLUS_EXPR, counter_i, counter_i, integer_one_node);
gimple_seq_add_stmt(&gseq, gstmt);
new_bb = create_basic_block(gseq, NULL, bb);
new_bb->discriminator = bb->discriminator + 1;
e = make_edge(bb, new_bb, EDGE_FALLTHRU);
set_immediate_dominator(CDI_DOMINATORS, new_bb, bb);
bb = new_bb;
then_bb = new_bb;
for(gsi = gsi_start_bb(bb); !gsi_end_p(gsi); gsi_next(&gsi))
    gimple_set_bb(gsi_stmt(gsi), bb);


/* insert if...cond: if i < 10 */
gseq = gimple_seq_alloc();
gstmt = gimple_build_cond(LT_EXPR, counter_i, build_int_cst(NULL_TREE, 10), NULL_TREE,
    NULL_TREE);
gimple_seq_add_stmt(&gseq, gstmt);
new_bb = create_basic_block(gseq, NULL, bb);
new_bb->discriminator = bb->discriminator + 1;
e = make_edge(bb, new_bb, EDGE_FALLTHRU);
set_immediate_dominator(CDI_DOMINATORS, new_bb, bb);
bb = new_bb;
cond_bb = new_bb;
for(gsi = gsi_start_bb(bb); !gsi_end_p(gsi); gsi_next(&gsi))
    gimple_set_bb(gsi_stmt(gsi), bb);

e = make_edge(cond_bb, then_bb, EDGE_TRUE_VALUE);
e->goto_locus = cfun->function_start_locus;
e->goto_block = DECL_INITIAL(current_function_decl);

e = make_edge(cond_bb, else_bb, EDGE_FALSE_VALUE);
e->goto_locus = cfun->function_start_locus;
e->goto_block = DECL_INITIAL(current_function_decl);

redirect_immediate_dominators(CDI_DOMINATORS, start_bb, bb);
set_immediate_dominator(CDI_DOMINATOR, first_new_bb, start_bb);
```

Figure 9: Instrmentation code for the figure 8

9 Mistakes in GCC Internal Manual

9.1 Traverse basic blocks

`bi` is no longer supported. Use `FOR_EACH_BB` to traverse each basic block, and use the `gsi` method mentioned in the manual to traverse the gimple statements in each basic block instead.