A tiny GCC front end – Part 2

Jan 6, 2016 • Roger Ferrer Ibáñez • compilers, GCC • gcc, tiny

The previous installment of this series was all about the syntax and the semantics of the tiny language. In this chapter we will start implementing a front end for tiny in GCC. The journey will be long but rewarding. Let's get started.

GCC

GCC is that ubiquitous compiler available in most UNIXs but mainly in GNU/Linux distributions. GCC is a big and old project, it started in 1987, and has been used as the system C compiler for many systems. Currently it supports several other programming languages including C++, Fortran, Ada and Go.

GCC has been written mostly in C but now some parts of it are (slowly) being migrated to C++. The compiler itself is one of these parts. This means that in its current state gcc is probably 80% C code compiled using C++. The dialect of C++ that can be used is C++03.

Initial setup

Since we are going to do development on GCC we will not use the source from a release but from their repository. GCC still uses Subversion for their development which is a bit unwieldy for local tracking of changes. Fortunately they have a read-only git mirror. This will do for local development.

Let's clone the gcc repository into a `gcc-src` directory. This directory is the *source tree*. This will take a while but only has to be done once.

```
$ git clone git://gcc.gnu.org/git/gcc.git gcc-src
Cloning into 'gcc-src'...
```

The next step is make sure we fulfill the requirements of GCC. The easiest way to do this is invoking a script found inside the `contrib` directory. Run it from the top level directory of the source tree.

```
$ cd gcc-src
$ ./contrib/download_prerequisites
... downloading stuff ...
```

This will add many files that ideally you want git to ignore them. In my case I added the following lines to the existing `gcc-src/.gitignore`.

```
gmp
gmp-4.3.2
gmp-4.3.2.tar.bz2
isl
isl-0.15
isl-0.15.tar.bz2
mpc
mpc-0.8.1
mpc-0.8.1.tar.gz
mpfr
mpfr-2.4.2
mpfr-2.4.2.tar.bz2
```

Let's create a branch and switch to it, where we will develop the tiny frontend.

```
$ git checkout -b tiny
Switched to a new branch 'tiny'
```

Now create a directory sibling to that of gcc, we will use it to build gcc. This directory is the *build tree*.

```
$ cd  ..           # leave source tree
$ mkdir gcc-build
```

Now let's configure a minimal gcc with just C and C++ (C++ is required for GCC itself).

```
$ cd gcc-build
$ ../gcc-src/configure --prefix=$(pwd)/../gcc-install --enable-languages=c,c++
```

And make an initial build of the whole GCC. This step may take several minutes depending on your specific machine. The flag to `-jN` will use all the cpus of your system.

```
$ make -j$(getconf _NPROCESSORS_ONLN)
... tons of gibberish ...
```

Finally let's install it.

```
$ make install
```

The compiler will be installed in a directory `gcc-install`, as a sibling of `gcc` and `gcc-build`.

Structure of GCC

GCC is huge. Period.

You may not be used to handle big projects. Ok, don't get scared. There are tools to help you. From full fledged IDEs like Eclipse to simpler (yet effective) tools like `ctags`. Use them!

In the source tree (`gcc-src`) we will find several directories. The most interesting one for us is `gcc` (i.e. `gcc-src/gcc`). The other directories are supporting libraries for gcc itself or runtime libraries required to run programs created with gcc (for instance `libgomp` or `libasan`). We are not going to use them, except, maybe `libcpp`. `libcpp` is mainly used to implement the C/C++ preprocessor in gcc but also provides location tracking support in gcc, more on this in another post. The GCC internals manual has the full list.

There are a few more directories in `gcc-src/gcc`. Directory `config` contains all the target-specific bits. In gcc *target* means «the environment for which we are generating code». In `config` you will find one subdirectory for architecture supported. If you are interested in this part of the compiler you may want to check `config/moxie`, it is small enough for a newcomer. Do not forget to check their great blog.

There is also one directory per language supported in `gcc-src/gcc`: `c` (C), `cp` (C++), `fortran`, `go`, `java`, `jit` (libgccjit), `lto` (Link Time Optimization), `objc` (Objective-C) and `objcp` (Objective-C++). Some of these frontends are not real programming languages (like jit or lto). They are *front ends* in the sense of *inputs* to the compiler: `libgccjit` uses as input the result of calling a JIT library, `lto` uses as input the streamed-to-disk intermediate representation of GCC, etc. There is also a `c-family` directory that contains common parts of C, C++, Objective-C and Objective-C++. Like before, the full list can be found in the GCC internals manual.

Adding a new front end is just a matter of creating a new directory in `gcc-src/gcc`. Do not worry if this stuff seems complex at first, there are plenty of other front ends that can be read as an example. In particular the `jit` and `go` front ends are relatively simple to be used as examples. Let's get down to it.

Initial boilerplate

We first need to create a `tiny` directory inside `gcc-src/gcc`. All our files will go there. no file outside of it will be changed.

```
$ cd gcc-src/gcc
$ mkdir tiny
```

The next step is telling GCC configure that we are going to build GCC with tiny support. This will fail. Do not worry, this is expected.

```
$ cd gcc-build
$ ../gcc-src/configure --prefix=$(pwd)/../gcc-install --enable-languages=c,c++,tiny
...
The following requested languages could not be built: tiny
Supported languages are: c,c,c++,fortran,go,java,jit,lto,objc,obj-c++
```

This is because GCC does not expect to have all the front ends available in a source tree. Rather than downloading the whole code of a release, you can download the gcc base and then add extra languages if you want.

Now, before we can proceed we will have to add some more files in `gcc-src/gcc/tiny`.

First we will add a `config-lang.in` file. This is a fragment of configure script. This file names the language (tiny in our case) and sets the name of the compiler (more on this below). It also specifies which languages are required to compile this front end. In our case we will use C++, so the command line option `--enable-languages` will require `c++` if we want to build `tiny`.

```
# gcc-src/gcc/config/config-lang.in
language="tiny"

compilers="tiny1\$(exeext)"

target_libs=""

gtfiles="\$(srcdir)/tiny/tiny1.cc"

# We will write the tiny FE in C++
lang_requires_boot_languages=c++

# Do not build by default
build_by_default="no"
```

Option `compilers` is the name of the compiler. Why is that? Because `gcc` is just a driver that internally calls the real compiler that will compile our code. Our *real compiler* will be called `tiny1` (the suffix 1 is due to historical reasons in the UNIX tradition). Option `gtfiles` is used to specify which files have to be scanned for the GCC own garbage collector mechanism. We will not use much of this for the moment.

Another file that we will need is `lang-specs.h`. This is a fragment of C header file. This file tells the gcc driver how and when to invoke the tiny1 compiler. In our case we want that files ended with `.tiny` are compiled with tiny1. These two lines will do. Just believe me here. If you want to understand what is going on, you can find more information in the file `gcc-src/gcc/gcc.c` and in GCC manual about spec files.

```
/* gcc-src/gcc/config/lang-specs.in */
{".tiny",   "@tiny", 0, 1, 0},
{"@tiny",   "tiny1 %i %(cc1_options) %{!fsyntax-only:%(invoke_as)}", 0, 1, 0},
```

The first line redirects `.tiny` files to `@tiny` specification. The second file states that tiny1 has to be invoked with the input file, `%i`. The next option states to use the content of variable `cc1_options`, `%(cc1_options)`. This is actually for the C compiler, but it has lots of useful defaults that will be handy for tiny. For instance it will make sure optimitzation options like `-Ox` and generic options like `-fXXX` are passed if specified. Finally if the user did not specify `-fsyntax-only`, we will invoke the assembler in order to generate the object, `%{!fsyntax-only:%(invoke_as)}`. Both variables `cc1_options` and `invoke_as` are defined in `gcc-src/gcc/gcc.c`. In particular `cc1_options` is probably overkill for tiny, but this way we avoid for now having to write our own.

A third file that will be required is `Make-lang.in`. This is another fragment of Makefile and will be used by the Makefile in `gcc-src/gcc` to build the tiny frontend. This file is a bit longer because it has to implement several goals. There is a first group of goals related to the driver (more on this below) and tiny1 and a second set, much larger, related to the frontend directory. Goals in this second group are of the form `tiny.`*`target`*.

Recall that `gcc` is the generic driver of GCC and when passed a `.tiny` file will invoke tiny1. This would work. But we want a `gcctiny` driver (similar to `gcc`, `g++`, `gfortran`) specific of our language. We only have to write a very small file for our gcctiny driver, the rest of the code is shared among drivers.

```
GCCTINY_INSTALL_NAME := $(shell echo gcctiny|sed '$(program_transform_name)')
GCCTINY_TARGET_INSTALL_NAME := $(target_noncanonical)-$(shell echo gcctiny|sed '$

tiny: tiny1$(exeext)

.PHONY: tiny

# Driver

GCCTINY_OBJS = \
   $(GCC_OBJS) \
   tiny/tinyspec.o \
   $(END)

gcctiny$(exeext): $(GCCTINY_OBJS) $(EXTRA_GCC_OBJS) libcommon-target.a $(LIBDEPS)
        +$(LINKER) $(ALL_LINKERFLAGS) $(LDFLAGS) -o $@ \
          $(GCCTINY_OBJS) $(EXTRA_GCC_OBJS) libcommon-target.a \
          $(EXTRA_GCC_LIBS) $(LIBS)

# The compiler proper

tiny_OBJS = \
    tiny/tiny1.o \
    $(END)

tiny1$(exeext): attribs.o $(tiny_OBJS) $(BACKEND) $(LIBDEPS)
        +$(LLINKER) $(ALL_LINKERFLAGS) $(LDFLAGS) -o $@ \
              attribs.o $(tiny_OBJS) $(BACKEND) $(LIBS) $(BACKENDLIBS)

tiny.all.cross:

tiny.start.encap: gcctiny$(exeext)
tiny.rest.encap:

tiny.install-common: installdirs
        -rm -f $(DESTDIR)$(bindir)/$(GCCTINY_INSTALL_NAME)$(exeext)
        $(INSTALL_PROGRAM) gcctiny$(exeext) $(DESTDIR)$(bindir)/$(GCCTINY_INSTALL
        rm -f $(DESTDIR)$(bindir)/$(GCCTINY_TARGET_INSTALL_NAME)$(exeext); \
        ( cd $(DESTDIR)$(bindir) && \
      $(LN) $(GCCTINY_INSTALL_NAME)$(exeext) $(GCCTINY_TARGET_INSTALL_NAME)$(exee

# Required goals, they still do nothing
```

```
43 tiny.install-man:
44 tiny.install-info:
45 tiny.install-pdf:
46 tiny.install-plugin:
47 tiny.install-html:
48 tiny.info:
49 tiny.dvi:
50 tiny.pdf:
51 tiny.html:
52 tiny.man:
53 tiny.mostlyclean:
54 tiny.clean:
55 tiny.distclean:
56 tiny.maintainer-clean:
57
58 # make uninstall
59 tiny.uninstall:
60         -rm -f gcctiny$(exeext) tiny1$(exeext)
61         -rm -f $(tiny_OBJS)
62
63 # Used for handling bootstrap
64 tiny.stage1: stage1-start
65         -mv tiny/*$(objext) stage1/tiny
66 tiny.stage2: stage2-start
67         -mv tiny/*$(objext) stage2/tiny
68 tiny.stage3: stage3-start
69         -mv tiny/*$(objext) stage3/tiny
70 tiny.stage4: stage4-start
71         -mv tiny/*$(objext) stage4/tiny
72 tiny.stageprofile: stageprofile-start
73         -mv tiny/*$(objext) stageprofile/tiny
74 tiny.stagefeedback: stagefeedback-start
75         -mv tiny/*$(objext) stagefeedback/tiny
```

Lines 1 and 2 define two variables that take the string gcctiny and apply some sed transformation that is kept in the Makefile and determined at configure time. This is used only for cross compilers so it is of little importance now. This will be used during install. In addition of installing gcctiny, a *target*-gcctiny will be installed as well. If you have x86-64 machine it will probably be something like `x86_64-pc-linux-gnu-gcctiny`.

Line 4 is a Makefile rule that says that the tiny goal requires building `tiny1$(exeext)`. `exeext` is a Makefile variable that the configure sets as empty in Linux but it is set to `.exe` in Windows, you will see it used everywhere a binary is mentioned.

Lines 8 to 19 are related to our gcctiny driver. Lines 10 to 13 we specify all the `.o` files required to build gcctiny. We list them in a variable called `GCCTINY_OBJS`. `GCC_OBJS` is a variable from `gcc-src/gcc/Makefile` that contains all the `.o` files required by gcc. This set is not complete to get a driver. So we add a `tinyspec.o` extra with a few definitions inside. More on this later. Lines 15 to 18 are the link command to build our gcctiny driver. No need to mess with that one, it works fine and most front ends use a similar command.

Lines 20 to 29 are related to tiny1. The real compiler. We follow a similar structure here. `tiny_OBJS` is a list of `.o` files of our compiler. Due to the way the makefile in `gcc-src/gcc` works, this variable has to be called *`lang`*`_OBJS` (in our case *`lang`* is tiny). Lines 26 to 28 are the link command to link tiny1. Again another command line taken from existing front ends that seems to work fine. No need to mess with that one either.

Now come a bunch of rules some of them do nothing, some of them do something. In line 35, this rule installs the `gcctiny` driver and makes a (hard) link to *`target`*`-gcctiny` in `bindir`. In this rule, variable `INSTALL_PROGRAM` is the `install` program (used obviously to install files), variable `bindir` is `gcc-install/bin`. The variable $(DESTDIR) is used only during `make install` to, temporarily, install files into another location before moving them to the final location (this is mostly useful for sysadmins and system packagers). Most of the time `DESTDIR` will be empty. Lines 59 to 61 implement the uninstall rule, that is invoked if during `make uninstall`. Finally lines 63 to 75 implement some logic required for the gcc bootstraping.

Great, we are half way. Now we need some code. Our current `Make-lang.in` mentions two files `tinyspec.o` and `tiny1.o` that have to be generated somehow. We will have to provide a `tinyspec.cc` and a `tiny1.cc`.

`tinyspec.cc` has to implement two functions and a variable.

```
void
lang_specific_driver (struct cl_decoded_option ** /* in_decoded_options */,
                      unsigned int * /* in_decoded_options_count */,
                      int * /*in_added_libraries */)
{
}

/* Called before linking.  Returns 0 on success and -1 on failure.  */
int
lang_specific_pre_link (void)
{
  /* Not used for Tiny.  */
  return 0;
}

/* Number of extra output files that lang_specific_pre_link may generate.  */
int lang_specific_extra_outfiles = 0; /* Not used for Tiny.  */
```

Some front ends may require changing the flags before they are passed to the driver. This is what the function `lang_specific_driver`. In our case it will do nothing because we do not have to change anything. So we will leave it empty. Function `lang_specific_pre_link` is called right before linking and can be used to do some extra steps and abort if they fail. This is not our case either. Finally the variable `lang_specific_extra_outfiles` is required to add some extra outfiles in the linking step. Only the Java front end seems to need this. We do not need it either, so it will be left as zero.

Finally, tiny1.cc. This is a rather big file full of boilerplate that we are not in position to fully understand yet. So just trust me here.

```
#include "config.h"
#include "system.h"
#include "coretypes.h"
#include "target.h"
#include "tree.h"
#include "gimple-expr.h"
#include "diagnostic.h"
#include "opts.h"
#include "fold-const.h"
#include "gimplify.h"
#include "stor-layout.h"
#include "debug.h"
#include "convert.h"
#include "langhooks.h"
#include "langhooks-def.h"
#include "common/common-target.h"

/* Language-dependent contents of a type.  */

struct GTY (()) lang_type
{
  char dummy;
};

/* Language-dependent contents of a decl.  */

struct GTY (()) lang_decl
{
  char dummy;
};

/* Language-dependent contents of an identifier.  This must include a
   tree_identifier.  */

struct GTY (()) lang_identifier
{
  struct tree_identifier common;
};

/* The resulting tree type.  */

union GTY ((desc ("TREE_CODE (&%h.generic) == IDENTIFIER_NODE"),
            chain_next ("CODE_CONTAINS_STRUCT (TREE_CODE (&%h.generic), "
                        "TS_COMMON) ? ((union lang_tree_node *) TREE_CHAIN "
                        "(&%h.generic)) : NULL"))) lang_tree_node
{
  union tree_node GTY ((tag ("0"), desc ("tree_node_structure (&%h)"))) generic;
  struct lang_identifier GTY ((tag ("1"))) identifier;
};

/* We don't use language_function.  */

```

```
struct GTY (()) language_function
{
  int dummy;
};

/* Language hooks.  */

static bool
tiny_langhook_init (void)
{
  /* NOTE: Newer versions of GCC use only:
           build_common_tree_nodes (false);
     See Eugene's comment in the comments section. */
  build_common_tree_nodes (false, false);

  /* I don't know why this has to be done explicitly.  */
  void_list_node = build_tree_list (NULL_TREE, void_type_node);

  build_common_builtin_nodes ();

  return true;
}

static void
tiny_langhook_parse_file (void)
{
  fprintf(stderr, "Hello gcctiny!\n");
}

static tree
tiny_langhook_type_for_mode (enum machine_mode mode, int unsignedp)
{
  if (mode == TYPE_MODE (float_type_node))
    return float_type_node;

  if (mode == TYPE_MODE (double_type_node))
    return double_type_node;

  if (mode == TYPE_MODE (intQI_type_node))
    return unsignedp ? unsigned_intQI_type_node : intQI_type_node;
  if (mode == TYPE_MODE (intHI_type_node))
    return unsignedp ? unsigned_intHI_type_node : intHI_type_node;
  if (mode == TYPE_MODE (intSI_type_node))
    return unsignedp ? unsigned_intSI_type_node : intSI_type_node;
  if (mode == TYPE_MODE (intDI_type_node))
    return unsignedp ? unsigned_intDI_type_node : intDI_type_node;
  if (mode == TYPE_MODE (intTI_type_node))
    return unsignedp ? unsigned_intTI_type_node : intTI_type_node;

  if (mode == TYPE_MODE (integer_type_node))
    return unsignedp ? unsigned_type_node : integer_type_node;

  if (mode == TYPE_MODE (long_integer_type_node))
    return unsignedp ? long_unsigned_type_node : long_integer_type_node;

  if (mode == TYPE_MODE (long_long_integer_type_node))
    return unsignedp ? long_long_unsigned_type_node
                     : long_long_integer_type_node;

  if (COMPLEX_MODE_P (mode))
    {
      if (mode == TYPE_MODE (complex_float_type_node))
        return complex_float_type_node;
      if (mode == TYPE_MODE (complex_double_type_node))
        return complex_double_type_node;
      if (mode == TYPE_MODE (complex_long_double_type_node))
        return complex_long_double_type_node;
      if (mode == TYPE_MODE (complex_integer_type_node) && !unsignedp)
        return complex_integer_type_node;
    }

  /* gcc_unreachable */
  return NULL;
}

static tree
tiny_langhook_type_for_size (unsigned int bits ATTRIBUTE_UNUSED,
                             int unsignedp ATTRIBUTE_UNUSED)
{
  gcc_unreachable ();
  return NULL;
}

/* Record a builtin function.  We just ignore builtin functions.  */

static tree
tiny_langhook_builtin_function (tree decl)
{
  return decl;
}

static bool
tiny_langhook_global_bindings_p (void)
{
  gcc_unreachable ();
  return true;
}

static tree
tiny_langhook_pushdecl (tree decl ATTRIBUTE_UNUSED)
{
  gcc_unreachable ();
}

```

```
157 static tree
158 tiny_langhook_getdecls (void)
159 {
160   return NULL;
161 }
162
163 #undef LANG_HOOKS_NAME
164 #define LANG_HOOKS_NAME "Tiny"
165
166 #undef LANG_HOOKS_INIT
167 #define LANG_HOOKS_INIT tiny_langhook_init
168
169 #undef LANG_HOOKS_PARSE_FILE
170 #define LANG_HOOKS_PARSE_FILE tiny_langhook_parse_file
171
172 #undef LANG_HOOKS_TYPE_FOR_MODE
173 #define LANG_HOOKS_TYPE_FOR_MODE tiny_langhook_type_for_mode
174
175 #undef LANG_HOOKS_TYPE_FOR_SIZE
176 #define LANG_HOOKS_TYPE_FOR_SIZE tiny_langhook_type_for_size
177
178 #undef LANG_HOOKS_BUILTIN_FUNCTION
179 #define LANG_HOOKS_BUILTIN_FUNCTION tiny_langhook_builtin_function
180
181 #undef LANG_HOOKS_GLOBAL_BINDINGS_P
182 #define LANG_HOOKS_GLOBAL_BINDINGS_P tiny_langhook_global_bindings_p
183
184 #undef LANG_HOOKS_PUSHDECL
185 #define LANG_HOOKS_PUSHDECL tiny_langhook_pushdecl
186
187 #undef LANG_HOOKS_GETDECLS
188 #define LANG_HOOKS_GETDECLS tiny_langhook_getdecls
189
190 struct lang_hooks lang_hooks = LANG_HOOKS_INITIALIZER;
191
192 #include "gt-tiny-tiny1.h"
193 #include "gtype-tiny.h"
```

That is a lot of stuff. First a bunch of includes that will be necessary. There is a bit of chaos in gcc headers, so it make take some tries until one figures the right list and its order of includes. Then language dependent definitions come, we need none of them, so they are almost empty. The `GTY (())` mark is used for the GCC garbage collector, we can ignore that for now.

Thsi file includes a number of `language hooks`. Language hooks are functions that can be overriden by the front end in order to implement language specific behaviour. Due to the C heritage of GCC this is implemented using macros. In line 187 the variable lang_hooks contains a LANG_HOOKS_INITIALIZER which in turn expands all the LANG_HOOKS_x of GCC. GCC provides default language hooks (defined in `langhooks.c` and described in `langhooks.h`). We can override them by undefining the associated macro and defining it to our specific function. Here we see some sensible defaults. If they fall short for some reason, we can always extend them at a later point.

Compilation of files starts by calling the hook `LANG_HOOKS_PARSE_FILE`. Or current code just prints a greeting and nothing else, see line 76. It will be enough to verify if things are working so far.

At the end of the file we include two extra headers `gt-tiny-tiny1.h` and `gtype-tiny.h` that have the routines automatically generated for the GCC garbage collector. If you recall the variable `gtfiles` in `config-lang.in` above, that variable mentions `tiny1.cc`. A tool called `gengtype` scans the files in `gtfiles` and using those `GTY` marks generates two headers with some functions that we have to include. The GCC internal manual has more information about the memory management.

Current layout

Our `gcc-src/gcc/tiny` directory now looks like this.

```
gcc-src/gcc/tiny
├── config-lang.in
├── lang-specs.h
├── Make-lang.in
├── tiny1.cc
└── tinyspec.cc
```

Hello gcctiny

By default gcc bootstraps itself. This means that gcc is compiled three times, in three steps called stages. In stage1 the system compiler is used. In stage2 the compiler compiled in stage1 is used to compile gcc. Likewise in stage3 the compiler compiled in stage2 is used to compile gcc. Assuming that the system compiler works correctly, all the objects generated in stage2 and stage3 should be identical. This is actually verified during a bootstrap. This is an excellent way to early detect problems in the compiler, but slows down development. This is why we will disable it when developing the front end. When we test the compiler we can reenable it again.

Now we can try again with the configure but this time we will disable the bootstrap, using `--disable-bootstrap`.

```
$ cd gcc-build
$ ../gcc-src/configure --prefix=$(pwd)/../gcc-install --disable-bootstrap --enable-la
$ make -j$(getconf _NPROCESSORS_ONLN)
... tons of gibberish ...
$ make install
```

A gcctiny and its corresponding target should now be in `gcc-install/bin`.

```
$ ls -1 gcc-install/bin/*tiny*
gcc-install/bin/gcctiny
gcc-install/bin/x86_64-pc-linux-gnu-gcctiny
```

Nice. Let's make a smoke test. First let's create an empty `test.tiny`. We need this because the driver checks for the existence of the input file for us.

```
$ touch test.tiny
$ gcc-install/bin/gcctiny -c test.tiny
Hello gcctiny!
```

Yay! I have passed the flag `-c` to avoid linking otherwise we would get an undefined error since there is no main function yet.

```
$ gcc-install/bin/gcctiny  test.tiny
Hello gcctiny!
/usr/lib/x86_64-linux-gnu/crt1.o: In function `_start':
(.text+0x20): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

If you want to see what is going on, just pass `-v`.

```
 1 $ gcc-install/bin/gcctiny -c -v test.tiny
 2 Using built-in specs.
 3 COLLECT_GCC=gcc-install/bin/gcctiny
 4 Target: x86_64-pc-linux-gnu
 5 Configured with: ../gcc-src/configure --prefix=/home/roger/soft/gcc/gcc-blog/gcc-
 6 Thread model: posix
 7 gcc version 6.0.0 20160105 (experimental) (GCC)
 8 COLLECT_GCC_OPTIONS='-c' '-v' '-mtune=generic' '-march=x86-64'
 9  /home/roger/soft/gcc/gcc-blog/gcc-install/bin/../libexec/gcc/x86_64-pc-linux-gnu
10 Tiny (GCC) version 6.0.0 20160105 (experimental) (x86_64-pc-linux-gnu)
11         compiled by GNU C version 5.3.1 20151219, GMP version 4.3.2, MPFR version
12 GGC heuristics: --param ggc-min-expand=30 --param ggc-min-heapsize=4096
13 Tiny (GCC) version 6.0.0 20160105 (experimental) (x86_64-pc-linux-gnu)
14         compiled by GNU C version 5.3.1 20151219, GMP version 4.3.2, MPFR version
15 GGC heuristics: --param ggc-min-expand=30 --param ggc-min-heapsize=4096
16 Hello gcctiny!
17 COLLECT_GCC_OPTIONS='-c' '-v' '-mtune=generic' '-march=x86-64'
18  as -v --64 -o test.o /tmp/ccsptWhB.s
19 GNU assembler version 2.25.90 (x86_64-linux-gnu) using BFD version (GNU Binutils
20 COMPILER_PATH=/home/roger/soft/gcc/gcc-blog/gcc-install/bin/../libexec/gcc/x86_64
21 LIBRARY_PATH=/home/roger/soft/gcc/gcc-blog/gcc-install/bin/../lib/gcc/x86_64-pc-l
22 COLLECT_GCC_OPTIONS='-c' '-v' '-mtune=generic' '-march=x86-64'
```

In line 9 tiny1 is being called. You can see some extra flags that are added because of `cc1_options` used in the `lang-specs.h`. In line 18 the assembler is invoked to generate the `.o` file. Since our frontend did nothing but print a message (line 16), the net effect is the same as compiling an empty file.

Wrap-up

We have now completed a basic step for our tiny front end. So we can start doing real work with it but this will be in the next chapter. That's all for today.

« A tiny GCC front end – Part 1 A tiny GCC front end – Part 3 »