GCC源码分析(四) — 语法/语义分析之声明说明符的解析

在gcc中词法分析是作为接口函数供语法分析调用的, 每当语法分析需要一个新的语法符号时其内部则会调用词法分析接口来获取一个新的token.
语法分析 = >语义分析后会将源代码转换成一个一个AST树节点, 之后此AST树节点即可代表整个源码中的所有内容了, 在gcc中函数c_common_parse_file则负责语法分析直到AST树结点的生成:

```
/*
toplev::main
 => do_compile
   => compile_file
     => lang_hooks.parse_file = c_common_parse_file()
*/
void c_common_parse_file (void)
{
    push_file_scope ();     // 创建file_scope,以记录编译单元中解析出的所有声明
    c_parse_file ();        // 解析整个编译单元,整个编译单元是由一个个外部声明组成的,每个外部声明解析后通常都会产生一个声明节点(decl), 此节点会被记录在file
    pop_file_scope ();      // 销毁当前file_scope
}
```

gcc中 **最大的编译单位是一个编译单元(translation-unit),一个编译单元也就是一个文件**, 而 **文件的构成元素则是一个个的外部声明(external-declaration),c语言的源代码实际上就是由一条条外部声明构成的**,其产生式如下:

```
translation-unit:
     external-declarations

external-declarations:
     external-declaration
     external-declarations external-declaration
```

在c代码中看到的不论是函数定义,变量定义,汇编或其他定义实际上都属于 **外部声明**,而如#include这样的则会被词法分析预处理,其不属于产生式的一部分,在解析产生式遇到时则自动被展开。
外部声明可以有5种形式,分别是 **函数定义(function-definition),声明(declaration),汇编定义(asm-definition),空语句(;)和extension扩展**,其产生式如下:

```
external-declaration:
     function-definition
     declaration

GNU extensions:
external-declaration:
     asm-definition
     ;
     __extension__ external-declaration
```

一般最常见的就是声明和函数定义,如 int x;就是一个声明, int func(..) {...} 就是一个函数定义,二者的共同点就是开头都是一个 **声明说明符(declaration-specifiers)**加一个 **声明符(declarator),** 需要注意的是声明和函数定义不是只出现在外部声明中的, 函数体内部的声明也是根据此产生式递归处理的,如:

```
int const * x, func(int); 中:
//int const 是一个声明说明符,其中每一个元素都叫一个说明符(xxx_specifier)(声明说明符的含义是声明符的说明符)
//*x和 func(int)都是一个声明符,其中每一个元素(如 *, x, func)都是一个c声明符(其都分别由一个c_declarator结构体表示的,这里称为c声明符)
```

在gcc源码解析中,每一个声明说明符都用一个 **c_declspecs**结构体表示, **此一个结构体中可以记录此声明说明符中所有说明符的信息**,而 **声明符则**不是用一个结构体表示的,而 **是用一个c_declarator的链表表示的**, 如 *x 中实际上是一个代表*的c_declarator链接一个代表x的c_declarator来记录此声明符的.
gcc **解析外部声明最终的目的(不论函数定义还是声明)就是要为此外部声明生成一个声明节点(_DECL),**而构建声明节点的函数(grokdeclarator)需要的输入主要就是前面的 **声明说明符(declaration-specifiers)和 声明符(declarator).**
在gcc中声明和函数定义(由于产生式开头相同)是由同一个函数c_parser_declaration_or_fndef来解析的，其调用路径为:

```
toplev::main
=> do_compile
   => compile_file
     => lang_hooks.parse_file = c_common_parse_file()
       => c_parse_file ();
         => c_parser_translation_unit (the_parser);     //解析编译单元
           => while()
              c_parser_external_declaration (parser);   //解析外部声明
                => c_parser_declaration_or_fndef (parser, true, true, true, false, true, NULL, vNULL);    //解析声明或函数定义
```

此函数最终会为声明或函数定义构建一个声明节点(XXX_DECL)并push到当前scope(如file_scope)中,但对于二者来说还有些不同

- 对于声明:
  - 根据声明说明符和声明符创建声明节点后,若此声明有初值(等号=)则会解析声明的初值
  - 若一个声明符解析完毕后,后面跟着逗号(,),则会循环解析下一个声明符,之后利用之前相同的声明说明符构建新的声明节点(DECL)
- 对于函数定义:
  - **函数定义和声明是互斥的**,如果前面只要走过了声明特有的流程,后续就不能按照函数定义来解析
  - 函数定义的特点是其后面有{}包裹的一个复合语句作为函数体(**没有{}的叫函数声明,也属于声明**),一个函数定义解析完毕后后面是不会跟着另一个函数定义的(不会循环解析)

二、声明说明符的解析

由前面可知,函数定义和声明都是以声明说明符开始的, 在gcc中二者解析是在同一函数c_parser_declaration_or_fndef中进行的, **c_parser_declaration_or_fndef函数中会先通过调用以下两个函数解析声明说明符**:

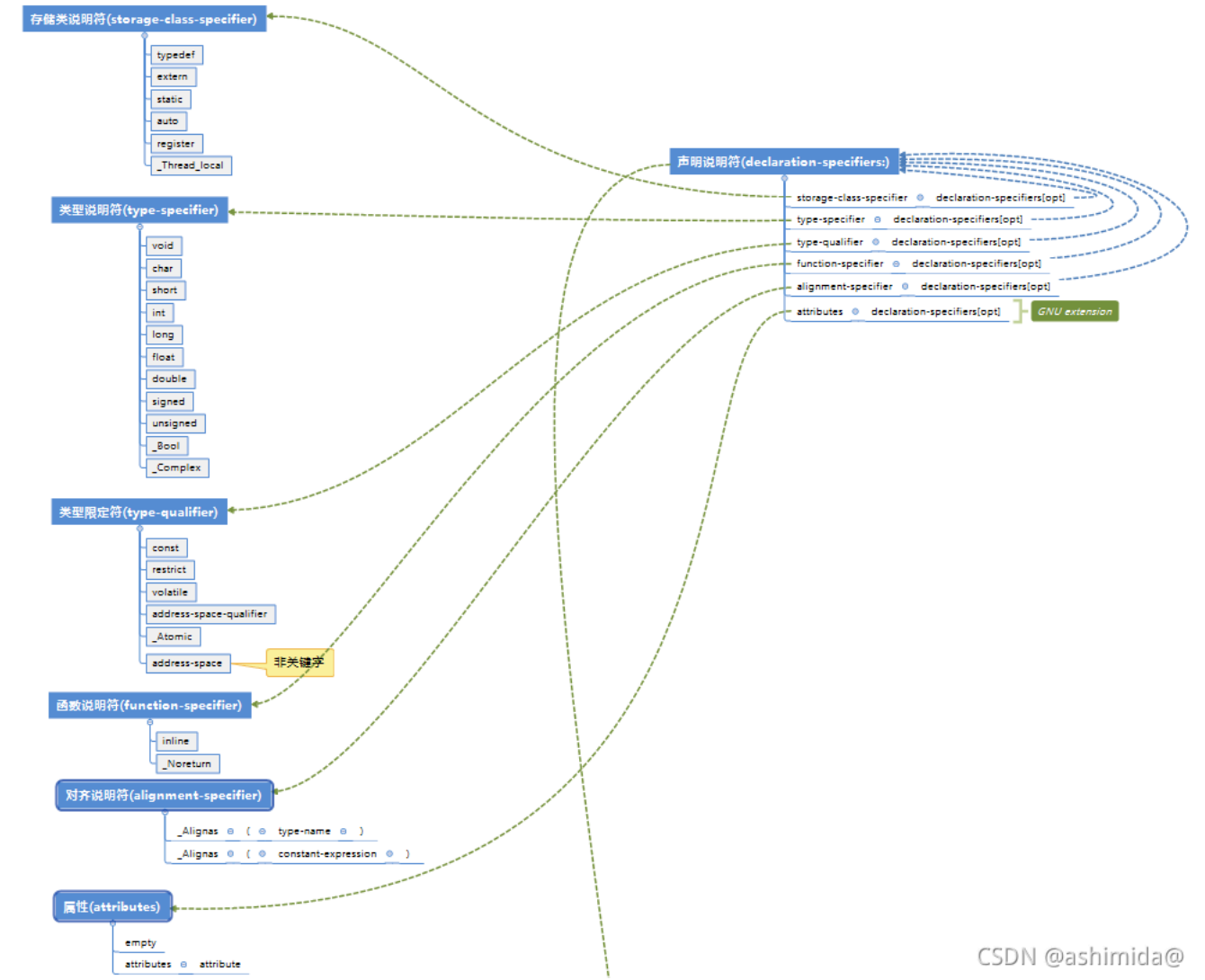
```
c_parser_declspecs (parser, specs, true, true, start_attr_ok,true, true, cla_nonabstract_decl);
finish_declspecs (specs);
```

声明说明符的解析过程实际上就是这两个函数的执行过程，其中主要的解析流程在c_parse_declspecs函数中进行。

2.1 声明说明符解析流程概述

在gcc中结构体struct c_declspecs记录整个声明说明符的所有信息，虽然是用一个结构体记录的，但实际上从语法符号来看声明说明符是可以由多个符号组成的，如上面的声明说明符static int; const char实际上都分别是由两个符号组成的，声明说明符中的每一个符号都称作一个说明符(specifier)，如上面的static, int, const, char分别都是一个说明符。

c_parse_declspecs函数前的注释给出了声明说明符的产生式,总结如下图:



注释中给出的这个产生式实际上和代码并不是完全相符的(大体上是相同的），主要有两点原因:

- 此产生式是有左递归的，没法用作自左向右的推导
- 实际代码中很多类型的说明符只能出现一个，而产生式中并没有体现出来

故后面声明说明符的分析过程此产生式只做参考， 并不完全匹配。

根据此产生式可知，声明说明符是由一个个**说明符(specifier)**构成的，这些说明符被分为5个不同的大类，分别是:

- 存储类说明符(**storage-class-specifier**)
- 类型说明符(**type-specifier**)
- 类型限定符(**type-qualifier**)
- 函数说明符(**function-specifier**)
- 对齐说明符(**alignment-specifier**)
- 属性(**attributes**)

而**c_parse_declspec函数的作用就是顺序分析语法符号，依次解析出此6类说明符并将说明符的信息记录到struct c_declspecs结构体中并返回**，此函数主体逻辑并不复杂，但涉及的细节较多代码较长，这里简化其代码逻辑，具体细节可参考源码:

```
1. void c_parser_declspecs (c_parser *parser, struct c_declspecs *specs,
2.
3.             bool scspec_ok, bool typespec_ok, bool start_attr_ok,
4.
5.             bool alignspec_ok, bool auto_type_ok,
6.
7.             enum c_lookahead_kind la)
8.
9. {
```

```
    /* 此函数基本就一个while循环，解析接下来遇到的字符串，如果是标识符或者是关键字则都有可能是说明符，
       判断如果是说明符则将信息记录到 specs(记录整个声明说明符的信息)中，不是则解析完毕。 */
    while (c_parser_next_token_is (parser, CPP_NAME)         //当前预读的是一个普通标识符
     || c_parser_next_token_is (parser, CPP_KEYWORD) ...
    {
       /*  基本的说明符大多数都是保留字，不会是普通标识符，但用户自定义的类型(如 x是 typedef int x;出来的，
           那么此时出现了代码 x A; 此时的x就不是保留字，而是一个普通标识符此外还有一个address_space不是保留字，这里就是分析这两种例外的 */
       if (c_parser_next_token_is (parser, CPP_NAME))

       {
          if (kind == C_ID_ADDRSPACE)         /* 解析类型限定符 address_space */

          {
             declspecs_add_addrspace (name_token->location, specs, as);
             /*  当前解析一个说明符成功，那么continue继续解析下一个说明符，所有保证循环继续的都代表成功的解析出了一个说明符，
               解析说明符失败则代表声明说明符解析完毕，后面就应该是声明符列表了  */
             continue;
          }
         /* 如果此标识符是一个用户自定义类型的类型说明符(如上面的x) */
         if (kind == C_ID_TYPENAME...
         {
            t.spec = lookup_name (value);        /* 获取类型说明符的类型声明节点 */
         }
         declspecs_add_type (name_token->location, specs, t);     /* 将类型说明符信息添加到声明说明符中 */
         continue;
       }
       /* 上面是说明符可能是普通标识符的情况，下面是最常见的说明符是关键字的情况，关键字的case比较多， 这里仅以关键字类型作为区分 */
       switch (c_parser_peek_token (parser)->keyword) {
          case 存储类说明符，函数说明符:
              declspecs_add_scspec (loc, specs,c_parser_peek_token (parser)->value);        /* 在声明说明符中记录此存储类说明符信息 */
              c_parser_consume_token (parser); break;            /* 消耗符号，循环继续 */
          case 类型说明符:
              /* 将类型信息暂存到一个struct c_typespec 节点t 中 然后将类型说明符节点t的所有信息都加入到声明说明符specs中 */
              declspecs_add_type (loc, specs, t);

          case 类型限定符:
              declspecs_add_qual (loc, specs, value);            /* 将类型限定符信息添加到 声明说明符 specs中 */

          case 属性:
              declspecs_add_attrs (loc, specs, attrs);            /* 将属性信息添加到声明说明符 specs中 */
          case 对齐:
              declspecs_add_alignas (loc, specs, align);          /* 将对齐信息添加到声明说明符 specs中 */

          case __GIMPLE/__RTL:                                    /* 对gimple和rtl函数的单独处理，这是个调试接口 */
       }
    }
}
```

根据上面的代码逻辑的简单梳理，可知整个声明说明符的分析过程，实际上就是确定当前语法符号(c_token)是属于哪类的哪个说明符，然后将此说明符信息添加到声明说明符的唯一结构体 c_declspecs specs中，此函数中具体有6个说明符处理函数,针对不同的类型,分别为:

```
1. declspecs_add_addrspace
2. declspecs_add_qual
3. declspecs_add_type
4. declspecs_add_scspec
5. declspecs_add_attrs
6. declspecs_add_alignas
```

此6个函数会具体的影响c_declspecs specs中的各个flag，若想了解c_declspecs各个字段的作用，则需要具体分析此6个函数。

这里需要注意的是,实际上**c_parser_declspecs并不只是作为声明说明符的解析函数，后面在声明解析时类型限定符列表(type-qualifier-list)也是用此函数解析的，因为二者的产生式形式基本相同**，区别仅在于后者只接受属性和类型限定符，只需要设置不同的c_parser_declspecs参数即可。

2.2 声明说明符结构体

声明说明符结构体各个字段的作用如下:

```
/*
   一个声明的声明说明符中可能有多个说明符,如 static int x; 中 static 和int分别是两个说明符，static int为此声明的声明说明符，
  而一个c_declspecs结构体就记录一个声明说明符中所有说明符的信息。
*/
struct c_declspecs {
  /*
     一个声明说明符可能由多个说明符构成，在解析源码过程中，每个说明符在源码中的位置都会记录到locations数组对应，
    locations数组中为大部分说明符都预留了单独的保存位置的字段，但对于互斥的说明符则仅仅保留了一个字段(如类型说明符只会出现一个，位置信息记录到cdw_typespec中
    存储类说明符记录到cdw_storage_class中)
     需要注意的是，位置信息在词法分析之前是只绑定到cpp_token/c_token上的(标识符节点上没有location信息)，词法分析之后便不会在使用这两个结构体了，
    所以在解析的过程中是需要在对应结构体中再次记录源码位置的。
    */
  location_t locations[cdw_number_of_elements];

  /* 若type有值，则代表当前声明说明符解析过程中已经确定了类型声明符,type记录此声明说明符的类型声明节点(TYPE_DECL)  */
  tree type;
  /* 对于typeof定义的声明,在代码执行过程中需要先执行此表达式来确定真正的类型，如int x; typeof(x) y; 对于其他的声明(其他类型说明符)，此字段无效 */
```

```
  tree expr;
  /* The attributes from a typedef decl.  */
  /* 对于typedef 自定义的类型，这里记录typedef自定义此类型时赋予给此类型的所有属性,对于其他的声明(其他类型说明符)，此字段无效 */
  tree decl_attr;
  /* 记录此声明中属性说明符指定的所有属性的链表 */
  tree attrs;
  /* 开始编译__GIMPLE或 __RTL函数的pass,见 c_parser_gimple_or_rtl_pass_list */
  char *gimple_or_rtl_pass;
  /* The base-2 log of the greatest alignment required by an _Alignas specifier, in bytes, or -1 if no such specifiers with nonzero alignment.  *
  int align_log;
  /* For the __intN declspec, this stores the index into the int_n_* arrays.  */
  int int_n_idx;
  /* For the _FloatN and _FloatNx declspec, this stores the index into the floatn_nx_types array.  */
  int floatn_nx_idx;
  /*
     记录当前声明说明符中使用了哪个存储类说明符，存储类说明符时互斥的，所以用了一个枚举类型代表具体哪个，
    如csc_auto,csc_extern,csc_register,csc_static,csc_typedef，见 declspecs_add_scspec
    */
  enum c_storage_class storage_class;
  /*
     在声明说明符推导过程中(c_parser_declspecs)如果遇到是用户自定义的类型说明符，则直接将此类型说明符的类型声明节点设置到 type字段即可，
    而若是遇到保留字的类型说明符，那么实际上只设置 typespec_word,直到最终finish_declspecs阶段才会设置真正的type(因为保留字中可能多个
    说明符才真正确定一个类型说明符，如unsigned long，所以在推导阶段此字段和type实际上是互斥的。
  */
  ENUM_BITFIELD (c_typespec_keyword) typespec_word : 8;
  /*
     若解析声明过程中解析到了类型说明符，则会将类型说明符的分配记录到这里，如ctsk_resword代表类型说明符是关键字;ctsk_typedef代表是用户自定义的类型(通过typec
     实际上此字段就是对类型说明符的一个细分。
  */
  ENUM_BITFIELD (c_typespec_kind) typespec_kind : 3;
  /* 代表是否发现了gimple或rtl保留字  */
  ENUM_BITFIELD (c_declspec_il) declspec_il : 3;
  /* 记录typeof的表达式expr是否可用于常量表达式 */
  BOOL_BITFIELD expr_const_operands : 1;
  /* Whether any declaration specifiers have been seen at all.  */
  /* 一个声明说明符中可能包含多个说明符，这个字段代表是否已经解析到至少一个说明符 */
  BOOL_BITFIELD declspecs_seen_p : 1;
  /* 当前声明说明符解析过程中，是否已经解析了非存储类说明符 */
  BOOL_BITFIELD non_sc_seen_p : 1;
  /* Whether the type is specified by a typedef or typeof name.  */
  /* 当前声明说明符解析过程中，是否已经解析处了 typedef 定义的用户自定义类型说明符 */
  BOOL_BITFIELD typedef_p : 1;
  /* Whether the type is explicitly "signed" or specified by a typedef
     whose type is explicitly "signed".  */
  BOOL_BITFIELD explicit_signed_p : 1;
  /* Whether the specifiers include a deprecated typedef.  */
  BOOL_BITFIELD deprecated_p : 1;
  /* Whether the type defaulted to "int" because there were no type
     specifiers.  */
  BOOL_BITFIELD default_int_p : 1;
  /* 代表当前声明说明符已经识别到一个long了  */
  BOOL_BITFIELD long_p : 1;
  /* 当前声明说明符已经识别到两个 long 标识符了 */
  BOOL_BITFIELD long_long_p : 1;
  /* 当前声明说明符之前已经识别到一个short了 */
  BOOL_BITFIELD short_p : 1;
  /* Whether "signed" was specified.  */
  /* 已经识别到了一个 signed */
  BOOL_BITFIELD signed_p : 1;
  /* Whether "unsigned" was specified.  */
  BOOL_BITFIELD unsigned_p : 1;
  /* Whether "complex" was specified.  */
  /* complex代表复数 */
  BOOL_BITFIELD complex_p : 1;
  /* Whether "inline" was specified.  */
  BOOL_BITFIELD inline_p : 1;
  /* Whether "_Noreturn" was speciied.  */
  BOOL_BITFIELD noreturn_p : 1;
  /* Whether "__thread" or "_Thread_local" was specified.  */
  BOOL_BITFIELD thread_p : 1;
  /* Whether "__thread" rather than "_Thread_local" was specified.  */
  BOOL_BITFIELD thread_gnu_p : 1;
  /* Whether "const" was specified.  */
  BOOL_BITFIELD const_p : 1;
  /* Whether "volatile" was specified.  */
  BOOL_BITFIELD volatile_p : 1;
  /* Whether "restrict" was specified.  */
  BOOL_BITFIELD restrict_p : 1;
  /* Whether "_Atomic" was specified.  */
  BOOL_BITFIELD atomic_p : 1;
  /* Whether "_Sat" was specified.  */
  BOOL_BITFIELD saturating_p : 1;
  /* Whether any alignment specifier (even with zero alignment) was specified.  */
  BOOL_BITFIELD alignas_p : 1;
  /* The address space that the declaration belongs to.  */
  /* 记录当前声明说明符的地址空间标号,实际上是当前 0-15的一个值，见 c_parser_declspecs */
  addr_space_t address_space;
};
```

声明说明符的这些字段只会在声明说明符分析过程中(c_parser_declspecs)被以上6个函数修改，以及在声明说明符分析结束时(finish_declspecs)修改.

2.3 类型说明符的解析

此分析过程同样是在c_parser_declspecs函数中，前面已经大体介绍过c_parser_declspecs函数的流程，这里只是摘取其中某些具体类型说明符分析的详细代码，这些case的共同特点都是要构建一个类型说明符(c_typespec)，然后调用declspecs_add_type将信息添加到c_declspecs中。

1.用户自定义的类型说明符

用户自定义的类型说明符在前面大体流程流程中已经有所体现，这里重新摘抄:

```
        if (kind == C_ID_TYPENAME...
         {
            /* 获取类型说明符的类型声明节点 */
            t.spec = lookup_name (value);
         }
         /* 将类型说明符信息添加到声明说明符中 */
         declspecs_add_type (name_token->location, specs, t);
```

2.普通类型保留字

这里包括下面的case都是在总体流程中类型说明符的细分，这里所谓的普通保留字包括:

```
    case RID_AUTO_TYPE:
      if (!auto_type_ok)
        goto out;
      /* Fall through.  */
    case RID_UNSIGNED:
    case RID_LONG:
    case RID_SHORT:
    case RID_SIGNED:
    case RID_COMPLEX:
    case RID_INT:
    case RID_CHAR:
    case RID_FLOAT:
    case RID_DOUBLE:
    case RID_VOID:
    case RID_DFLOAT32:
    case RID_DFLOAT64:
    case RID_DFLOAT128:
    // CASE_RID_FLOATN_NX:
    case RID_BOOL:
    case RID_FRACT:
    case RID_ACCUM:
    case RID_SAT:
    case RID_INT_N_0:
    case RID_INT_N_1:
    case RID_INT_N_2:
    case RID_INT_N_3:
          ......
      /* 代表当前声明说明符的类型说明符为一个保留字(关键字)，如 int */
      t.kind = ctsk_resword;
      /* 对于保留字(CPP_KEYWORD)，其spec节点为c_token->value,也就是标识符的lang_identifier节点 */
      t.spec = c_parser_peek_token (parser)->value;
      t.expr = NULL_TREE;
      t.expr_const_operands = true;

      declspecs_add_type (loc, specs, t);
      /* 解析完毕，消耗掉此token */
      c_parser_consume_token (parser);
      break;
```

3.枚举类型

枚举类型有个专门的子函数c_parser_enum_specifier来构建类型说明符，如下:

```
    case RID_ENUM:
      if (!typespec_ok)
        goto out;
      attrs_ok = true;
      seen_type = true;
      t = c_parser_enum_specifier (parser);
          invoke_plugin_callbacks (PLUGIN_FINISH_TYPE, t.spec);       //这里还有个plugin的callback
      declspecs_add_type (loc, specs, t);
      break;
```

4.结构体与联合体

结构体和联合体同样有个专门构建类型说明符的子函数c_parser_struct_or_union_specifier，如下:

```
    case RID_STRUCT:
    case RID_UNION:
      if (!typespec_ok)
        goto out;
      attrs_ok = true;
      seen_type = true;
      t = c_parser_struct_or_union_specifier (parser);
          invoke_plugin_callbacks (PLUGIN_FINISH_TYPE, t.spec);        //这里还有个plugin的callback
```

```
      declspecs_add_type (loc, specs, t);
      break;
```

5.typeof

typeof同样有个专门构建类型说明符的子函数c_parser_typeof_specifier,如下:

```
    case RID_TYPEOF:
      if (!typespec_ok || seen_type)
        goto out;
      attrs_ok = true;
      seen_type = true;
      t = c_parser_typeof_specifier (parser);
      declspecs_add_type (loc, specs, t);
      break;
```

声明说明符在解析到类型说明符时会构造一个类型说明符结构体并传给declspecs_add_type函数，实际上生成此结构体的目的是为了在declspecs_add_type上少传几个参数，和其他的declspecs_add_*函数不同的是，类型说明符需要设置的参数较多，所以才有了struct c_typespec这么一个类型说明符的结构体:

```
## ./gcc/c/c-tree.h
struct c_typespec {
  /* 记录当前类型说明符的类型，如是使用了关键字还是自定义类型，最终体现到specs->typespec_kind字段 */
  enum c_typespec_kind kind;
  /* 代表的是 expr中的表达式是否适用于常量表达式，最终体现到specs->typespec_kind字段 */
  bool expr_const_operands;
  /* The specifier itself.  */
  /*
    此结构体指向此类型说明符自身的声明节点
    * 对于用户自定义(typedef定义)的类型说明符(kind = ctsk_typedef),spec指向此类型说明符的声明节点(TYPE_DECL),若类型未定义则指向error_mark_node
      后面declspecs_add_type函数中会设置specs->type = TYPE_TYPE(spec)，也就是说specs->type指向此声明节点的类型节点，代表用户自定义的类型。
    * 对于使用系统内置的(保留字中的)类型说明符(kind = ctsk_resword,如 int),spec指向此类型说明符的标识符节点(lang_identifier)
      可能因为关键字的声明是全局的，而自定义类型是要看scope的，所以对于自定义类型就直接获取了对应的类型声明节点，而对于关键字直接拿标识符节点即。
      对于保留字，实际上影响的是specs->typespec_word字段，最后在finish_declspecs函数中才会转换为具体的type.
  */
  tree spec;
  /* 这个是在确定type类型之前要执行的表达式，如int x; typeof(x) r1，那么在确定r1类型之前，就要执行此表达式，除了typeof未见到其他使用位置 */
  tree expr;
};
```

2.4 存储类说明符、函数说明符的解析

此过程同样在c_parser_declspecs函数中，存储类说明符和函数说明符的解析流程都是一样的，如下:

```
    case RID_STATIC:
    case RID_EXTERN:
    case RID_REGISTER:
    case RID_TYPEDEF:
    case RID_INLINE:
    case RID_NORETURN:
    case RID_AUTO:
    case RID_THREAD:
      if (!scspec_ok)
        goto out;
      attrs_ok = true;
      declspecs_add_scspec (loc, specs, c_parser_peek_token (parser)->value);
      c_parser_consume_token (parser);
      break;
```

对于存储类说明符和函数说明符，实际上都是直接调用declspecs_add_scspec函数并传入此说明符的标识符节点，此函数就会将信息添加到声明说明符中。

2.5 类型限定符的解析

此过程同样在c_parse_declspecs函数中，除了atuomic稍微复杂，其他的也都比较简单:

```
    case RID_ATOMIC:
          . .....
      attrs_ok = true;
      tree value;
      value = c_parser_peek_token (parser)->value;
      c_parser_consume_token (parser);
      if (typespec_ok && c_parser_next_token_is (parser, CPP_OPEN_PAREN))
        {
          ......
          declspecs_add_type (loc, specs, t);
        }
      else
        declspecs_add_qual (loc, specs, value);
      break;
    case RID_CONST:
    case RID_VOLATILE:
    case RID_RESTRICT:
      attrs_ok = true;
```

```
      declspecs_add_qual (loc, specs, c_parser_peek_token (parser)->value);
      c_parser_consume_token (parser);
      break;
```

可以看到，对于类型限定符，除了atuomic可能调用declspecs_add_type函数外，通常都是传入标识符节点直接调用**declspecs_add_qual** 函数。

2.6 属性的解析

此过程同样在c_parse_declspecs函数中，如下:

```
    case RID_ATTRIBUTE:
      if (!attrs_ok)
        goto out;
      attrs = c_parser_attributes (parser);
      declspecs_add_attrs (loc, specs, attrs);
      break;
```

如果发现当前的保留字是__attribute__/__attribute，则先调用c_parse_attributes从源码中解析出一个属性，然后通过函数declspecs_add_attrs 将属性记录到声明说明符中.

2.7 对齐的解析

此过程同样在c_parse_declspecs函数中，如下:

```
    case RID_ALIGNAS:
      if (!alignspec_ok)
        goto out;
      align = c_parser_alignas_specifier (parser);
      declspecs_add_alignas (loc, specs, align);
      break;
```

2.8 GIMPLE/RTL解析

此过程同样在c_parse_declspecs函数中，声明说明符产生式中没有出现的两个例外就是__GIMPLE/__RTL关键字，二者实际上是用来调试gimple和rtl代码的，其流程如下:

```
    case RID_GIMPLE:    //"__GIMPLE 关键字
      if (! flag_gimple)
        error_at (loc, "%<__GIMPLE%> only valid with %<-fgimple%>");
      c_parser_consume_token (parser);
      specs->declspec_il = cdil_gimple;
      specs->locations[cdw_gimple] = loc;
      c_parser_gimple_or_rtl_pass_list (parser, specs);
      break;
    case RID_RTL:        // "__RTL" 关键字
      c_parser_consume_token (parser);
      specs->declspec_il = cdil_rtl;
      specs->locations[cdw_rtl] = loc;
      c_parser_gimple_or_rtl_pass_list (parser, specs);
      break;
```

2.9 声明说明符解析结束

声明说明符的解析流程为:

```
c_parser_declaration_or_fndef
 => c_parser_declspecs (parser, specs, true, true, start_attr_ok, true, true, cla_nonabstract_decl);
 => finish_declspecs (specs);
```

c_parser_declspecs结束，就代表从表达式上声明说明符的匹配已经结束了，最终调用的finish_declspecs函数对保留字的类型说明符设置具体的type节点(保留字在c_parser_declspecs只设置了typespec_word并没有设置type，这里专门负责将保留字的typespec_word对应到具体的type上；对于非保留字的类型说明符，这里什么也不做直接返回)