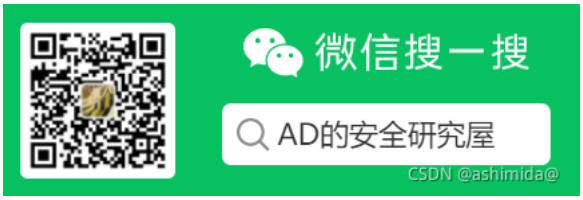
GCC源码分析(八) — 语法/语义分析之声明与函数定义的解析

更多内容可关注微信公众号



一、概述

由前可知, c_parser_declaration_or_fndef负责对声明和函数定义的解析, 对于**声明的解析其整体流程是**:

1. 解析声明说明符(c_parser_declspecs)
2. 解析出第一个声明符(c_parser_declarator)
3. 根据声明说明符 + 声明符为声明构建声明节点(XXX_DECL)，并将新创建的声明绑定到当前scope(start_decl)
4. 若下一个符号是等号(=)则解析此声明符的初值(c_parser_initializer); 不论是否有=,当前声明符的解析均到此结束了(finish_decl)
5. 若下一个符号是逗号(,)则继续回到步骤2)继续解析下一个声明符
6. 若下一个符号是冒号(;)则解析结束

对于**函数定义的解析其整体流程是**:

1. 解析声明说明符(c_parser_declspecs)
2. 解析出第一个声明符(c_parser_declarator)
3. 根据声明说明符 + 声明符为其构建声明节点(FUNCTION_DECL),将新创建的声明绑定到当前scope并创建新的scope解析函数体(start_function)
4. 在新的scope中绑定函数参数声明链表中的所有声明(store_parm_decls)
5. 解析整个函数体(c_parser_compound_statement)
6. 将函数体解析结果记录到函数声明节点中,同时对函数声明节点(中函数体的内容)执行AST->GENERIC
7. 函数定义解析完毕(由产生式可知其不会循环解析)

二、源码分析

由前可知一个编译单元(translation-unit)是由一句句的外部声明(external-declarations)组成的,外部声明一共有5种类型,分别是:

- **函数定义(function-definition)**
- **声明(declaration)**
- **汇编定义(asm-definition)**
- **分号结束(;)**
- **__extension + 外部声明**

其中最主要的就是函数定义(function-definition)和声明(declaration),二者的产生式如下:

```
function-definition:
     declaration-specifiers[opt] declarator declaration-list[opt] compound-statement

declaration:
     declaration-specifiers init-declarator-list[opt] ;
     ......

init-declarator-list:
     init-declarator
     init-declarator-list , init-declarator

init-declarator:
     declarator simple-asm-expr[opt] attributes[opt]
     declarator simple-asm-expr[opt] attributes[opt] = initializer
```

根据产生式可知**函数定义和声明都通常都是是以声明说明符(declaration-specifiers) + 声明符(declarator)开头的**(函数定义可省略声明说明符的特殊case先忽略),前面的分析中已经提及到:

- **声明说明符的解析流程**：声明说明符是由若干个语法元素组成的，其中的每个语法元素都称为一个说明符(*specifier)(如int const * x; 中int 和const分别是两个说明符,二者共同的组成了声明说明符), 在**gcc中所有只为声明说明符分配了一个结构体(c_declspecs)**,其中包含各种flag可以完全记录所有的说明符的出现.
- **声明符的解析流程**: 声明符也是由若干个语法元素组成的, 这些词法元素在产生式中没有统一的描述(可以是一个*,函数,数组,或标识符),但**在gcc源码中,每一个声明符中的词法元素都记录在一个c_declarator结构体中(后续称为c声明符)**, 此结构体组成的一个链表,就代表了整个声明符(如 int const * x; 解析过程中会分别为 * 和x生成两个c_declarator结构体, *的链接x的,最终返回*的c_declarator,也就是整个链表,代表当前的声明符)

概述中可知 c_parser_declaration_or_fndef 前两步的分析中解析到了一个声明说明符结构体(c_declspecs)和一个声明符链表结构体(c_declarator链表), 而在此之后函数定义(function-definition)和声明(declaration)的产生式就不同了:

- 对于**函数定义,其后面通常是直接跟符合语句({}包裹的一段语句)**
- 对于**声明(不论是变量还是函数声明),其后面通常跟=(代表赋值)或,(下一个声明)或;(声明结束)**

c_parser_declaration_or_fndef的整体流程简化如下:

```
static void
c_parser_declaration_or_fndef (c_parser *parser, ......)
{
    ......
    /* 解析出一个声明说明符 */
    specs = build_null_declspecs ();
    c_parser_declspecs (parser, specs, true, true, start_attr_ok, true, true, cla_nonabstract_decl);
    finish_declspecs (specs);

    /* 循环解析声明符并做处理 */
    while (true)
    {
        struct c_declarator *declarator;
        /* 解析出一个声明符, 最终结果是个c声明符的链表 */
        declarator = c_parser_declarator (parser, specs->typespec_kind != ctsk_none, C_DTR_NORMAL, &dummy);
        /* 匹配到等号(=),冒号(;),逗号(,) 都代表这是一个声明而不是函数定义 */
        if (c_parser_next_token_is (parser, CPP_EQ)
          || c_parser_next_token_is (parser, CPP_COMMA)
          || c_parser_next_token_is (parser, CPP_SEMICOLON)
          ......)
        {
            /* 若解析完声明符后出现了等号,则代表这是一个变量声明，且需要赋初值 */
            if (c_parser_next_token_is (parser, CPP_EQ))
    `       {
                ......
                /* 先根据声明说明符和声明符构建此声明的声明树节点,并push到当前scope */
                d = start_decl (declarator, specs, true, chainon (postfix_attrs, all_prefix_attrs));
                /* 解析变量初值,并设置到其声明节点中 */
                start_init (d, asm_name, global_bindings_p (), &richloc);
                init = c_parser_initializer (parser);
                finish_init ();
                /* finish_decl的作用可暂时理解为除了为声明设置初值节点外就是调用plugin的回调 */
                finish_decl (d, init_loc, init.value, init.original_type, asm_name);
            } else {
                /* 对于非等号的情况下，也同样是创建声明树节点，只不过此时不需要赋初值了 */
                tree d = start_decl (declarator, specs, false, chainon (postfix_attrs, all_prefix_attrs));
                finish_decl (d, UNKNOWN_LOCATION, NULL_TREE, NULL_TREE, asm_name);
            }

            /* 上面先处理了可能有等号(=)赋值的情况,因为当前肯定是声明,故继续判断是逗号(,)继续解析下一个声明符;还是冒号(;)解析结束 */
            if (c_parser_next_token_is (parser, CPP_COMMA))    continue;
            else if (c_parser_next_token_is (parser, CPP_SEMICOLON))  return;
            .....
        }

        /* 走到这里说明没有当前没有匹配到声明(包括函数声明),那么匹配到的必定是一个函数定义 */
        /* start_function和start_decl的作用类似,都是先为此声明创建一个声明树节点并push到当前scope,
           区别在于其会为后续函数体也创建scope,且会创建result节点等 */
        start_function (specs, declarator, all_prefix_attrs)
        /* 将函数形参中的所有声明都添加到函数体的scope中(同时形参中所有节点的上下文被设置为当前函数声明) */
        store_parm_decls ();
        /* 解析函数体内的所有语句，最终返回一个tree_statement_list */
        fnbody = c_parser_compound_statement (parser);
        /* 若函数体解析结果非空,则将结果添加到全局cur_stmt_list中 */
        if(fnbody) add_stmt (fnbody);
        /* 将全局cur_stmt_list中的所有语句解析结果构建成一个BIND_EXPR保存到函数声明中
           此函数同时负责将函数的AST->GENERIC，以及最终AST的gimplify流程 */
        finish_function ();
    }
}
```

不论是声明还是函数定义,在声明说明符+声明符解析完毕之后，二者均会先分别使用start_decl/start_function函数构建一个声明树节点并将其push到当前scope中,这两个函数的代码有很多类似之处，记录如下:

```
/* 此函数主要就是调用grokdeclarator为声明(如类型声明,变量声明,函数声明)创建了一个声明树节点，并将此声明树绑定到当前scope上了 */
tree start_decl (struct c_declarator *declarator, struct c_declspecs *declspecs,
        bool initialized, tree attributes)
{
    ......
```

```
  /* 为声明(如类型声明,变量声明,函数声明)创建声明节点 */
  tree decl = grokdeclarator (declarator, declspecs,
             NORMAL, initialized, NULL, &attributes, &expr, NULL, deprecated_state);
  /* 若创建声明节点失败，则返回 NULL_TREE */
  if (!decl || decl == error_mark_node)
    return NULL_TREE;

  if (initialized)    /* 若此声明需要初始化 */
    {
      /* 所有文件scope的声明,且有初值则都需要静态存储空间 */
      if (current_scope == file_scope)
        TREE_STATIC (decl) = 1;
      /* 初始化初值为 error_mark_node */
      DECL_INITIAL (decl) = error_mark_node;
    }

  /* 对于全局未初始化的非per thread变量，若没有指定no common flag，则标记此全局变量应该分配到.common段. */
  if (VAR_P (decl) && !initialized && TREE_PUBLIC (decl)
      && !DECL_THREAD_LOCAL_P (decl) && !flag_no_common)
    DECL_COMMON (decl) = 1;

  /* 将decl绑定到其标识符上,深度与current_scope相同,并设置此声明的上下文节点; 若同一个scope中重复定义,则要么去重(保留旧的并返回),要么报error; */
  tem = pushdecl (decl);

  /* 如外部声明赋初值了,则其被认为是一个非外部声明,且同样需要静态存储 */
  if (initialized && DECL_EXTERNAL (tem))
    {
      DECL_EXTERNAL (tem) = 0;
      TREE_STATIC (tem) = 1;
    }

  return tem;
}

/* 此函数只在处理函数定义时调用,其先调用grokdeclarator为函数定义创建一个声明树节点，然后同样将此声明去重后push到当前scope,
   但与start_decl不同的是,此函数还同时为函数体创建了一个新的scope,且为函数定义创建了一个返回值节点. */
bool start_function (struct c_declspecs *declspecs, struct c_declarator *declarator,
        tree attributes)
{
  tree decl1, old_decl;
  tree restype, resdecl;
  location_t loc;

  /* 此函数为函数定义创建一个声明树节点,需要注意的是,这里传入的参数是FUNCDEF代表函数定义，而声明传入的是NORMAL,二者的区别包括:
     * NORMAL时grokdeclarator会检查声明符是否满足函数定义的语法(需要有cdk_function类型的c声明符)
     * FUNCDEF时会在全局变量current_function_arg_info中记录参数的参数类型列表解析的结果(from c_arg_info) */
  decl1 = grokdeclarator (declarator, declspecs, FUNCDEF, true, NULL,
              &attributes, NULL, NULL, DEPRECATED_NORMAL);

  /* 在正式解析函数体之前,这里有一个plugin的回调函数 */
  invoke_plugin_callbacks (PLUGIN_START_PARSE_FUNCTION, decl1);

  /* 当前函数还没有设置current_function_decl指向自己的声明树节点，这里非空代表
    当前函数定义是在某个其他的函数定义内部的,则此函数定义标记为非public的
  */
  if (current_function_decl != NULL_TREE)
    TREE_PUBLIC (decl1) = 0;

  /* 将属性增加到函数声明节点 */
  c_decl_attributes (&decl1, attributes, 0);

  /* 标记函数声明的初值为error_mark_node，而不是NULL_TREE */
  DECL_INITIAL (decl1) = error_mark_node;

  /* 函数定义都是需要静态分配存储空间的 */
  TREE_STATIC (decl1) = 1;

  /* 将decl绑定到其标识符上,深度与current_scope相同,并设置此声明的上下文节点; 若同一个scope中重复定义,则要么去重(保留旧的并返回),要么报error;
     注意这里返回的声明节点直接记录到current_function_decl中了 */
  current_function_decl = pushdecl (decl1);

  /* 这里新起了一个scope,此时尚未设置next_is_function_body,是新起的scope,
    此scope用来记录函数参数类型列表中的所有声明,以及函数体语句中的所有内容 */
  push_scope ();

  /* 标记当前scope中记录有参数声明链表的声明 */
  declare_parm_level ();

  /* 获取函数的返回值类型 */
  restype = TREE_TYPE (TREE_TYPE (current_function_decl));
  /* 构建一个返回值声明节点，返回值声明节点不需要名字 */
  resdecl = build_decl (loc, RESULT_DECL, NULL_TREE, restype);
  /* 表示此返回值节点是编译器自动构建的 */
  DECL_ARTIFICIAL (resdecl) = 1;
  /* 表示此节点在符号表中应该被忽略 */
  DECL_IGNORED_P (resdecl) = 1;
  /* 在函数声明节点记录其返回值节点 */
  DECL_RESULT (current_function_decl) = resdecl;
  start_fname_decls ();
  /* 返回true代表函数声明创建并绑定成功 */
  return true;
}
```

start_decl和start_function的公共部分都是通过grokdeclarator函数创建一个声明,并通过pushdecl函数将此声明push到当前scope,这两个函数源码如下:

```
/*
  grokdeclarator函数可以翻译为摸索声明符的,其输入包括一个声明的声明说明符(declspecs)和声明符(declarator),二者决定了要声明的对象的类型和名字,
此函数最终会为此声明生成一个声明节点(_DECL)或一个类型节点(_TYPE,对于sizeof,cast)并返回,其大体逻辑如下:
  * 首先若当前声明说明符中的存储类声明符为typedef,那么此函数就会返回一个TYPE_DECL声明节点
  * 如果没有指定typedef限定符,那么此函数的返回节点就与其传入的参数decl_context相关:
    - 若decl_context = TYPENAME,则此函数返回一个类型节点(sizeof,cast)
    - 若decl_context = PARM,则此函数返回一个参数声明节点(PARM_DECL)
    - 若decl_context = FIELD,则此函数返回一个成员声明节点(FIELD_DECL,结构体,联合体,枚举型)
  * 若非以上场景,则如果最终解析后的类型是一个FUNCTION_TYPE,则此函数返回一个函数声明(FUNCTION_DECL)节点
  * 若非以上场景,则最终返回一个变量声明节点(VAR_DECL)
  注:
   1)上述的非以上场景,则decl_context只可能是 NORMAL 或 FUNCDEF,对于二者的流程大体一致,区别在于
     FUNCDEF只接受函数声明符,且最终会将参数类型列表记录到全局变量current_function_arg_info中
   2)指针和数组并不影响最终返回的声明节点的类型,其只影响声明节点的类型节点的类型
   3)此函数除了创建声明节点外,还会设置其各种属性, 如TREE_PUBLIC/TREE_STATIC/TREE_EXTERNAL等
*/
static tree grokdeclarator (const struct c_declarator *declarator, struct c_declspecs *declspecs,
        enum decl_context decl_context, bool initialized, tree *width, tree *decl_attrs, tree *expr, bool *expr_const_operands,
        enum deprecated_states deprecated_state)
{
  /* 声明说明符中确定的类型的树节点,可能是系统预置类型或之前解析typedef声明时新生成的用户自定义类型,
    注意这个类型未必是最终声明的类型,对于函数定义,数组,指针来说,声明说明符上的类型都不是其最终类型(type最后会记录其最终类型). */
  tree type = declspecs->type;
  /* 声明说明符中确定是哪个存储类说明符(互斥的) */
  enum c_storage_class storage_class = declspecs->storage_class;
  int type_quals = TYPE_UNQUALIFIED;    /* 代表当前指定了哪个类型限定符 */
  /* 记录当前声明符中, 最终的那个普通声明符的标识符树节点(lang_identifier), 如 int *p; 中p的标识符树节点. */
  tree name = NULL_TREE;
  /* 代表当前是否在为函数定义创建声明树节点,只有decl_context = FUNCDEF时后面会设置此flag */
  bool funcdef_flag = false;
  /* 代表当前声明符中是否满足一个函数声明符的语法(c_declarator链表中是否有cdk_function的节点) */
  bool funcdef_syntax = false;
    /* 对于函数声明符,记录其声明符分析过程中对参数类型列表的解析结果 */
  struct c_arg_info *arg_info = 0;
  /* 对于函数定义, 设置funcdef_flag=true,然后就按照NORMAL上下文处理 */
  if (decl_context == FUNCDEF)
    funcdef_flag = true, decl_context = NORMAL;

  /* 此block中用来确定当前声明符最终的标识符节点(=> name),以及当前声明符中是否满足函数声明符的语法(=> funcdef_syntax) */
  {
    /* 遍历整个c_declarator链表 */
    const struct c_declarator *decl = declarator;
    while (decl)
      switch (decl->kind)
    {
    case cdk_array:
      loc = decl->id_loc;    /* 如果遍历到了数组说明符, 那么说明符位置先记录下来 */
    case cdk_function:
    case cdk_pointer:        /* 碰到函数或指针c声明符 */
      funcdef_syntax = (decl->kind == cdk_function);    /* 若在遍历过程中发现了函数声明符(发现括号), 则标记funcdef_syntax为true */
      decl = decl->declarator;    /* 遍历下一级声明符 */
      break;
    case cdk_id:            /* 碰到普通声明符 */
      loc = decl->id_loc;   /* 记录位置 */
      if (decl->u.id)       /* 获取整个普通声明符的标识符树节点(lang_identifier) */
        name = decl->u.id;
      decl = 0;    /* 到普通声明符解析就结束了,普通声明符逻辑上就应该是整个声明符链上最后一个声明符 */
      break;
        case ...
    }
  }
  /* 若当前在解析函数定义,但声明符中没有匹配到函数声明符,则显然是语法错误, 直接返回NULL_TREE */
  if (funcdef_flag && !funcdef_syntax)
    return NULL_TREE;
  /* 整合声明说明符中所有的类型限定符 */
  type_quals = ((constp ? TYPE_QUAL_CONST : 0) | (restrictp ? TYPE_QUAL_RESTRICT : 0) ......

  /* 这里遍历整个声明符链表中的所有c声明符节点,直到遇到普通声明符(逻辑上有应该是最后一个c声明符)为止, 这个循环的目的就是为了确定当前声明符最终的类型:
     * 对于普通声明符(如 int x;)来说,其声明说明符中的类型就是其最终的类型, 不必做任何处理
     * 而对于指针,数组,函数声明符来说(如 int *y, p[], func(int n);)来说,声明说明符中的类型只是其指向节点/数组元素/函数返回值的类型,
       并不是此声明节点真正的类型,这个循环中需要依次解析c声明符列表,确定要生成的声明树节点的最终的类型.
     最终确定的类型同样记录在type中, 一开始的type是声明说明符的type,经过此循环就变为了声明树节点最终的type. */
  while (declarator && declarator->kind != cdk_id)
    {
      switch (declarator->kind)    /* 分别对属性,指针,数组,函数c声明符做处理 */
    {
    case cdk_attrs:    /* 如果发现了属性节点, 则全部串联到全局的returned_attrs中 */
      {
        ......
        returned_attrs = decl_attributes (&type, chainon (returned_attrs, attrs), attr_flags);
        break;
      }
    case cdk_array:    /* 对于数组, 函数, 指针 主要就是用来重新确定type */
      {
        ......
        type = build_array_type (type, itype);
```

```
        break;
      }
    case cdk_function:
      {
        arg_info = declarator->u.arg_info;    /* 获取函数c声明符中的的参数类型列表,或标识符列表 */
        /*  当前是在处理声明符,对于函数c声明符,当前的arg_info需要是一个参数类型类表,这里检查如果是参数类型链表,则返回
          参数声明的类型节点链表,如果是标识符链表则返回空 */
        arg_types = grokparms (arg_info, really_funcdef);
        /* 先根据类型限定符,为函数的返回值创建变种类型节点(若需要) */
        type = c_build_qualified_type (type, quals_used);
        /* 类型限定符的作用就是确定变种，确定后此类型限定符相当于使用完毕(清空) */
        type_quals = TYPE_UNQUALIFIED;
        /* 构建一个函数类型节点,这里type是前面处理过类型限定符(变种)的函数返回值类型; arg_types是参数类型的链表，若声明中出现了标识符链表，则这里arg_types为
           此函数最终生成一个类型为FUNCTION_TYPE的类型节点，作为此函数声明符的类型(若之前已创建过此类型节点,则返回之前的节点,通过节点hash来判断) */
        type = build_function_type (type, arg_types);
        /* 为当前函数c声明符生成类型节点,则当前c声明符的使命就完成了,继续处理下一个c声明符 */
        declarator = declarator->declarator;
        /* 若当前参数列表中有定义结构体,则设置结构体的上下文为函数类型节点 */
        FOR_EACH_VEC_SAFE_ELT_REVERSE (arg_info->tags, ix, tag)
          TYPE_CONTEXT (tag->type) = type;
        break;
      }
    case cdk_pointer:
      {
        /* 如果有类型限定符,同样先为类型创建变种 */
        if (type_quals)
          type = c_build_qualified_type (type, type_quals, orig_qual_type, orig_qual_indirect);
        /* 这里为类型type创建一个指向type的指针类型并返回 */
        type = c_build_pointer_type (type);
        /* 获取指针c声明符后面的类型限定符,此类型限定符要作用生成后的type上，通过这里理解 int const * p; const int *p; int * const p;的区别:
           * int const * p; 和 const int *p; 是一样的，因为 int const 不论什么顺序出现,其都是一个声明说明符,声明说明符的解析结果一样,
             其后面的声明符的内容也一样,所以二者的含义是一样的，按照这里的逻辑:
             - 首先有个类型 int; 然后为其做了个变种类型int const; 然后为这个变种类型做了个指针(POINTER_TYPE)作为变量p的类型
             - 也就是说变量p的本质是一个指针,其用来指向一个int const类型的节点.
             - 所以这里p叫做常量指针
           * int * const p; 逻辑为:
             - 首先有个类型int; 然后为此类型做了个指针(POINTER_TYPE); 然后还要为这个指针做个const的变种指针(也是POINTER_TYPE)作为变量p的类型
             - 也就是说变量p的本质是一个常量的指针,其值是固定不能改的,其指向的内容是一个int类型的节点.
             - 所以这里的p叫做指针常量*/
        type_quals = declarator->u.pointer_quals;
        /* 继续解析下一个c声明符 */
        declarator = declarator->declarator;
        break;
      }
    default:
      gcc_unreachable ();
    }
    }
  /* 到这里,则当前的c声明符则指向了最终的普通标识符节点 */
  /* 整合此声明在声明说明符和声明符中所有属性，最终记录到decl_attrs 中并返回给父函数 */
  *decl_attrs = chainon (returned_attrs, *decl_attrs);

  /* 如存储类限定符指定了typedef,则说明当前是一个类型定义,此函数直接返回此类型声明节点(TYPE_DECL) */
  if (storage_class == csc_typedef)
    {
      tree decl;
      /* 同样先为类型创建对应增加了类型限定符的变种 */
      if (type_quals)
        type = c_build_qualified_type (type, type_quals, orig_qual_type, orig_qual_indirect);
      /* 为typedef创建类型声明节点并返回 */
      decl = build_decl (declarator->id_loc, TYPE_DECL, declarator->u.id, type);
      return decl;
    }
  /* 对于sizeof, cast的情况,返回一个类型节点 */
  if (decl_context == TYPENAME)
    {
        type = c_build_qualified_type (type, type_quals, orig_qual_type, orig_qual_indirect);
        return type;
    }

  /* 非typedef的情况则在这里根据不同的case创建对应的声明(DECL)或类型(TYPE)节点 */
  {
    tree decl;
    if (decl_context == PARM)    /* 若当前在解析函数的参数列表,则走这里(若解析标识符类表不会走到此函数) */
    {
     tree promoted_type;
     bool array_parameter_p = false;

     if (TREE_CODE (type) == ARRAY_TYPE)    /* 若当前的类型为数组，举例就是 int func(int x[]); 中的 int x[]的解析 */
         ......
     else if (TREE_CODE (type) == FUNCTION_TYPE)    /* 若当前的类型为函数类型，如 int func(int func1(int x)); 中的 int func1(int x)*/
        type = c_build_pointer_type (type);         /* 对于参数类型类表中的函数声明,最终构造的声明节点的类型是一个函数类型的指针类型 */
     else if (type_quals)                           /* 非以上的两种情况,如果有类型限定符,则先根据最终的类型限定符为类型创建变种节点,如 int * const x; i
      type = c_build_qualified_type (type, type_quals);
     /* 这里是为当前参数声明创建最终的声明节点,对于参数声明不论最终的声明类型,一律返回一个PARM_DECL类型的声明节点 */
     decl = build_decl (declarator->id_loc, PARM_DECL, declarator->u.id, type);
     promoted_type = c_type_promotes_to (type);     /* 提升类型(若需要,如float默认变为double),作为调用此函数时为此声明传入的参数类型(实参) */
     DECL_ARG_TYPE (decl) = promoted_type;
    }
    else if (decl_context == FIELD)         /* 同样是直接解析类型限定符,不论最终是何种类型,一律返回一个FIELD_DECL类型的声明节点*/
    {
      ......
      type = c_build_qualified_type (type, type_quals, orig_qual_type, orig_qual_indirect);
```

```
      decl = build_decl (declarator->id_loc, FIELD_DECL, declarator->u.id, type);
    }
    else if (TREE_CODE (type) == FUNCTION_TYPE)
    {
      /* 若当前不是在解析参数类型列表(PARM),也不是在解析结构体内部的域(FILED),那么若当前最终解析出的节点的类型是函数类型(FUNCTION_TYPE),
        则这里为其构建一个函数声明节点(FUNCTION_DECL)并返回(若当前scope中有重复定义则会去重) */
      decl = build_decl (declarator->id_loc,FUNCTION_DECL, declarator->u.id, type);
      /* 将前面解析出的所有属性的链表记录到当前函数声明中 */
      decl = build_decl_attribute_variant (decl, decl_attr);
      /* 这里决定一个函数是否是外部函数包括:非file_scope的auto函数一定不是external的; extern inline的是否是external的跟是否设置了gnu89有关;
       其他函数只要是没有初值(正常函数声明),就认为是external的,如果有初值(也就是start_function来的函数定义),就认为是非external的(可以理解为
       函数有定义则就不是外部函数,没有定义(只是声明)的函数就默认认为是外部函数) */
      if (storage_class == csc_auto && current_scope != file_scope)
        DECL_EXTERNAL (decl) = 0;
      else if (declspecs->inline_p && storage_class != csc_static)
        DECL_EXTERNAL (decl) = ((storage_class == csc_extern) == flag_gnu89_inline);
      else
        DECL_EXTERNAL (decl) = !initialized;
      /* 除了static(当前编译单元可见)和auto的函数,其他函数都是编译单元外可见的 */
      TREE_PUBLIC (decl) = !(storage_class == csc_static || storage_class == csc_auto);
      /* 对于函数定义来说(有函数体), 记录其参数信息到全局变量 */
      if (funcdef_flag)
        current_function_arg_info = arg_info;
    }
    else
    {
      /* 若当前不是在解析参数类型列表(PARM),也不是在解析结构体内部的域(FILED),那么若当前最终解析出的节点的类型也不是函数类型(FUNCTION_TYPE),
        那么这里就会为当前声明创建变量声明节点(VAR_DECL)*/
      /* extern和初始化本身就是不能共存的,这里的extern_ref才代表这是一个真正的extern变量,其除了要求指定了extern外,还需是未初始化的,这样后续才会被真正设置为
      int extern_ref = !initialized && storage_class == csc_extern;
      /* 解析类型限定符 */
      type = c_build_qualified_type (type, type_quals, orig_qual_type, orig_qual_indirect);

      /*
        非函数,参数声明,结构体的声明都在这里构建,这里都构建了一个变量声明节点(VAR_DECL)
        如int const x; 这里也是构建了一个变量声明节点(而不是常量声明CONST_DECL节点),const只不过使其多了一个readonly属性而已
      */
      decl = build_decl (declarator->id_loc, VAR_DECL, declarator->u.id, type);
      /* 对于大多数只要带extern的,其就是个external变量了,问题无非在于extern+初值的情况,前面的检查代码(省略)中如果遇到非file_scope的extern+初值会直接报错,
        故到这里就排除了非file_scope的情况,对于file_scope的extern+初值的变量声明则会在start_decl函数中重置为非extern */
      DECL_EXTERNAL (decl) = (storage_class == csc_extern);
      if (current_scope == file_scope)
      {
        /*  在文件scope中,只要此声明没指定static说明符,那么默认此声明就是会标记为TREE_PUBLIC */
        TREE_PUBLIC (decl) = storage_class != csc_static;
        /* 在文件scope中,只要此声明真的不会被当做extern处理,就会标记为TREE_STATIC,这里的static代表要分配静态存储空间 */
        TREE_STATIC (decl) = !extern_ref;
      }
      else
      {
        /* 非文件scope的, 指定了static修饰符则代表要静态分配存储空间,标记TREE_STATIC */
        TREE_STATIC (decl) = (storage_class == csc_static);
        /* 非文件scope显示指定了extern,且没有初始化的,则都会认为是外部声明(前面标记过了),外部声明会则同时会被标记为TREE_PUBLIC
           在非文件scope, 只有外部声明会被标记为public,其他都不是public
         注: 这里没有单独为如函数内部的extern int x = 0;取消DECL_EXTERNAL的设置,因为此函数前面的检查代码(省略了)中对于非file_scope的extern+初值
           形式直接报错了, 只有file_scope中的extern+初值会当做非extern处理.*/
        TREE_PUBLIC (decl) = extern_ref;
      }
    /* register关键字的意思是代表建议将此变量存储在寄存器中 */
    if (storage_class == csc_register)
    {
       C_DECL_REGISTER (decl) = 1;
       DECL_REGISTER (decl) = 1;
    }
    /* 为类型限定符在声明树节点上设置对应的flag,如TREE_READONLY,SIDE_EFFECT等 */
    c_apply_type_quals_to_decl (type_quals, decl);

       /* 在声明节点上设置对齐 */
    if (alignas_align)
    {
       SET_DECL_ALIGN (decl, alignas_align * BITS_PER_UNIT);
       DECL_USER_ALIGN (decl) = 1;
    }
    return decl;    /* 返回新创建的声明节点 */
  }
}
```

```
/*
   此函数的输入参数是一个要绑定到当前scope的声明节点,此函数的主要作用是调用bind函数将声明x绑定到其标识符上(scope为current_scope),但与bind函数不同的是:
   1) 若当前声明不是在file_scope中的声明(也就是在函数体内),那么要为此声明设置上下文节点为其所在的函数体对应的函数声明节点(current_function_decl)
   2) 若当前scope中一个符号有重复的定义,则会检查重复的定义是否冲突;不冲突则将所有内容合并到原有定义中(duplicate_decls),如果冲突则返回错误.
      如函数声明后面又出现了函数定义的问题,二者的声明最终会融合为一个,但需要注意的是这里合并的是同一个scope中的重复定义,对于不同scope的同名定义,默认是子scope
   3) 若当前声明是file_scope中的声明,则要为此声明默认在external_scope做一个符号绑定,只不过此符号绑定是invisible的.
   此函数的返回值也是个声明节点,如果此声明的标识符在当前scope没有绑定过声明则直接返回声明节点x; 若之前有绑定过并通过检查,则返回的就是之前的那个声明节点(x则在
  的过程中释放掉.
*/
tree pushdecl (tree x)
{
  /* 获取此声明节点的标识符节点 */
  tree name = DECL_NAME (x);
```

```
  /* 获取当前scope，也就是此声明绑定的那个scope */
  struct c_scope *scope = current_scope;
  struct c_binding *b;
  /* current_function_decl非空代表当前在某个函数体内部解析,语法分析阶段只有start_function函数中会设置current_function_decl,
    也就意味着一定是进入到哪个函数体中了才会有current_function_decl. 任何不在文件scope中的声明都要有上下文节点,除非这是函数体内
    部的一个extern声明.这里给所有非真正extern的节点都设置上下文. */
  if (current_function_decl
      && (!VAR_OR_FUNCTION_DECL_P (x)
      || DECL_INITIAL (x) || !DECL_EXTERNAL (x)))
    DECL_CONTEXT (x) = current_function_decl;
  if (!name) {
      /* 匿名声明，直接绑定到当前scope中即可，绑定后就直接返回,如参数声明或结构体成员都可能有匿名声明 */
      bind (name, x, scope, /*invisible=*/false, /*nested=*/false, locus);
      return x;
  }

  /* 非匿名声明，则获取标识符节点上已有的符号绑定(symbol_binding) */
  b = I_SYMBOL_BINDING (name);
  /* 在绑定前确定此符号在当前scope下是否已经有了声明绑定,若有则属于重复定义,则需要尝试对重定义进行merge,不能merge的则要报error.
     一个标识符只有在同一个scope中有重定义才会尝试merge,不同scope的定义则是子scope的覆盖父scope. */
  if (b && B_IN_SCOPE (b, scope))
  {
      struct c_binding * b_use = b;
      /*
        这里尝试将同一scope重名的声明进行merge:
        * 若merge成功则返回ture，所有信息都记录在旧的声明中(b_use->decl),新声明(x)直接被释放.
        * 若merge失败则返回false
      */
      if (duplicate_decls (x, b_use->decl))
      {
        /* merge成功，新声明的所有信息都记录到旧的声明节点(b_use->decl)中,最终返回旧的声明 */
        return b_use->decl;
      }
      else
        goto skip_external_and_shadow_checks;
  }

  /* 到这里说明当前scope中之前没有对标识符name 绑定声明  */

  /* 这里为所有文件scope中声明都为其生成一个invisible的外部声明 */
  if (DECL_EXTERNAL (x) || scope == file_scope)
            if (TREE_PUBLIC (x))
        bind (name, x, external_scope, /*invisible=*/true, /*nested=*/false, locus);

skip_external_and_shadow_checks:
  /* 真正执行绑定操作,将当前声明绑定到当前scope中,返回的还是此声明节点x */
  bind (name, x, scope, /*invisible=*/false, nested, locus);
  return x;
}
```

以上的两个函数就是创建声明(grokdeclarator)并将声明绑定当当前scope(pushdecl)的基本流程,**当遇到";"则声明解析完毕, 而如果遇到"{"则会按照函数定义的流程解析**, 函数定义在start_function创建声明并绑定到当前scope后，还要处理函数体,后续主要依次涉及三个函数:

```
store_parm_decls ();     /* 将函数形参中的所有声明都添加到函数体的scope中 */

fnbody = c_parser_compound_statement (parser);    /* 解析函数体内的所有语句，最终返回一个BIND_EXPR节点 */
if(fnbody) add_stmt (fnbody);     /* 若函数体解析结果非空,则将此BIND_EXPR封装成一个statement_list_node,并添加到全局cur_stmt_list中 */

finish_function ();    /* 将全局cur_stmt_list中的所有语句解析结果构建成一个BIND_EXPR保存到函数声明中; 此函数同时负责将函数的AST->GENERIC */
```

这里主要介绍流程相关的store_parm_decls和finish_function函数:

```
/* 此函数将函数参数类型列表中的所有声明节点都绑定到当前的scope */
void store_parm_decls (void)
{
  /* 获取start_function中记录的函数声明符,只有函数定义才会记录,函数声明不记录 */
  tree fndecl = current_function_decl;
  bool proto;
  /* 同样获取start_function中记录的函数参数列表信息，获取后将其置零 */
  struct c_arg_info *arg_info = current_function_arg_info;
  current_function_arg_info = 0;

  /* 此函数将参数类型列表中的所有参数声明,常量,类型,结构体声明等都绑定到当前scope中,也就意味着这些声明后续对当前scope可见.
     同时将参数声明链表记录到fndecl->arguments上(参数声明记录到对应的函数声明节点中) */
    store_parm_decls_newstyle (fndecl, arg_info);

  /* 标记下一步要解析函数体了,标记此全局变量后后面push_scope则会使用当前已经记录了参数声明的整个scope */
  next_is_function_body = true;

  /* 为当前函数分配 function结构体，并初始化部分内容 */
  allocate_struct_function (fndecl, false);

  /* 初始化函数体节点,函数体节点要有一个新的 tree_statment_list (cur_stmt_list指向此节点) */
  DECL_SAVED_TREE (fndecl) = push_stmt_list ();
}
```

```
static void store_parm_decls_newstyle (tree fndecl, const struct c_arg_info *arg_info)
{
  tree decl;
  /* 这里循环遍历当前函数的参数类型链表,并将所有参数声明都绑定到当前scope, 这也是为什么在函数体内可以直接使用函数形参中的变量. */
  for (decl = arg_info->parms; decl; decl = DECL_CHAIN (decl))
    {
      /* 设置参数声明.context指向当前函数 */
      DECL_CONTEXT (decl) = current_function_decl;
      /* 将当前标识符声明和标识符节点绑定, 信息同步到标识符节点和当前scope */
      if (DECL_NAME (decl))
        bind (DECL_NAME (decl), decl, current_scope, /*invisible=*/false, /*nested=*/false, UNKNOWN_LOCATION);
      else
        /* 函数声明中的参数声明可以只有类型而没有变量名,但函数定义中的参数声明是必须有变量名的,故这里发现没有变量名会报错 */
        error_at (DECL_SOURCE_LOCATION (decl), "parameter name omitted");
    }

  /* 在函数声明中记录函数的参数声明链表 */
  DECL_ARGUMENTS (fndecl) = arg_info->parms

  /* 函数参数类型列表中也可能定义了类型声明和常量声明,将这些声明(若有),也都绑定到当前scope,且上下文指向当前函数 */
  for (decl = arg_info->others; decl; decl = DECL_CHAIN (decl))
  {
      DECL_CONTEXT (decl) = current_function_decl;
      if (DECL_NAME (decl))
        bind (DECL_NAME (decl), decl, current_scope, /*invisible=*/false, /*nested=*/(TREE_CODE (decl) == FUNCTION_DECL), UNKNOWN_LOCATION);
  }

  /* 将参数类型列表中所有的结构体声明也都绑定到当前scope */
  FOR_EACH_VEC_SAFE_ELT_REVERSE (arg_info->tags, ix, tag)
    if (tag->id)
      bind (tag->id, tag->type, current_scope, /*invisible=*/false, /*nested=*/false, UNKNOWN_LOCATION);
}
```

```
/* 将全局cur_stmt_list中的所有语句解析结果构建成一个BIND_EXPR保存到函数声明中; 此函数同时负责将函数的AST->GENERIC, 以及最终AST的gimplify流程 */
void finish_function (void)
{
  /* 获取当前正在处理的函数的声明节点 */
  tree fndecl = current_function_decl;
  /* 前面函数体解析过程中(c_parser_compound_statement)会push_scope然后pop_scope, 解析结果(tree_block)记录到DECL_INITIAL(current_function_decl)中
    也就是当前函数的initial节点中,故到这里函数正常是会有DECL_INITIAL的,其记录的是函数体中解析出的所有声明节点的信息(一个 tree_block) */
  if (DECL_INITIAL (fndecl) && DECL_INITIAL (fndecl) != error_mark_node)
    /* 设置函数体的block的上下文为此函数声明,之前pop_scope时应该已经设置过了(实测这里确实之前pop_scope时已经默认应该设置过了) */
    BLOCK_SUPERCONTEXT (DECL_INITIAL (fndecl)) = fndecl;

  /* 设置result节点的上下文也为当前函数, result节点时在start_function函数中创建的 */
  if (DECL_RESULT (fndecl) && DECL_RESULT (fndecl) != error_mark_node)
    DECL_CONTEXT (DECL_RESULT (fndecl)) = fndecl;
  /* 这里最终会返回一个BIND_EXPR代表函数体,细节后续描述 */
  DECL_SAVED_TREE (fndecl) = pop_stmt_list (DECL_SAVED_TREE (fndecl));
  /* 设置声明在elf符号表中是否可见 */
  c_determine_visibility (fndecl);

  /* 若当前函数定义有tree_block就会走这里, 只要有函数体的{}, 就一定会走这里 */
  if (DECL_INITIAL (fndecl) && DECL_INITIAL (fndecl) != error_mark_node && !undef_nested_function)
  {
      /* 对于全局函数定义来说到这里正常是还没有设置context的,总是会进这个if的 */
      if (!decl_function_context (fndecl))
      {
        /* 这里是AST转GENERIC树之前的hook点 */
        invoke_plugin_callbacks (PLUGIN_PRE_GENERICIZE, fndecl);
        /* C前端将AST转为 GENERIC, 这一步是对AST进行规范化处理,对于gcc c来说实际上基本不需要处理 */
        c_genericize (fndecl);
        /* 为当前函数创建cgraph_node */
        cgraph_node::finalize_function (fndecl, false);
        }
  }
  set_cfun (NULL);    /* 重置全局变量cfunc 为空 */
  /* 当前函数处理完毕后的回调,并清空current_function_decl */
  invoke_plugin_callbacks (PLUGIN_FINISH_PARSE_FUNCTION, current_function_decl);
  current_function_decl = NULL;
}
```

根据如上所述,函数定义的后续流程可以总结为:

1. 为函数定义创建函数声明树节点, 并将其绑定到当前scope; 创建新的scope解析函数体(start_function)
2. 在新的scope中先绑定函数参数声明链表都的所有声明(store_parm_decls)
3. 解析整个函数体(c_parser_compound_statement)
4. 最后将函数体解析结果记录到函数声明节点中,同时对函数声明节点(中函数体的内容)执行AST->GENERIC