GCC LALR Syntax (3)
C 언어 표현식에 대한 세부 처리 과정

정원교

2004년 5월 13일

목 차

제 1 절 23 주 문서를 시작하며

이번 문서에서는 22 주에서 언급했던 각 변수의 선언이라던가, 함수의 정의들은 GCC 에서 실제로 처리하게 될 때, 어떠한 operation 이 일어나는지에 대해서 살펴볼 것이다.

제 2 절 Type 의 정의

"Type 의 정의"라는 말부터 설명을 한다면, C 언어에서 변수를 선언할 때는 반드시 그에 맞는 해당 type 을 정의를 해주어야 한다. 그것은 int 혹은 float, double 등등이 될 수 있고, 앞에 register, static, extern 등등의 storage class specifier 들을 지정해 줄 수 있다. 물론 attribute 들도 이에 해당될 수 있는데, 이에 대해서는 함수도 마찬가지로 적용된다. 함수의 경우 그 함수가 처리가 완료된 후 반환할 return 값의 type 을 지정해 줄 수 있다. 그래서 실제로 모든 변수와 함수가 해석될 때는 대부분의 경우, type 이 먼저 오고, 다음에 변수의 이름 혹은 함수의 이름이 오기 때문에 Type 이 어떻게 정의되는지에 대해 살펴볼 필요가 있겠다하겠다.

Type 을 저장하게 되는 것은 $prefix/gcc/c-parse.in 파일에 정의되어 있는 4 개의 전역 변수를 통해서 STACK 형태로 관리가 되게 된다.

- current_declspecs

  현재 declaration 의 type 을 가지고 있다.

- prefix_attributes

  현재 declaration 에 정의되어 있는 prefix attribute 들을 가지고 있다.

- all_prefix_attributes

  현재 declaration 에 정의된 모든 attribute 들에 대한 list 를 가지고 있는데, 위에서 선언된 prefix_attributes 를 포함할 뿐만아니라, comma (,) 뒤에 올 수 있는 attribute 들 또한 모두 포함한다.

- declspec_stack

  앞에서 정의된 세개의 전역변수 current_declspecs 와 prefix_attributes, all_prefix_attributes 가 저장될 필요가 있을 경우, 여기에 TREE_LIST node 를 통해서 저장되게 된다.

GCC 에서는 좀 더 편리하게 여러개의 type 정의 관련 부분을 저장하기 위해 STACK 을 사용한다고 앞에서 언급하였다. 이러한 operation 을 수행하기 위해서, 두개의 macro 를 제공한다. 하나의 macro 의 경우, PUSH_DECLSPEC_STACK 이며, 세부 내용은 아래와 같다.

```
#define PUSH_DECLSPEC_STACK                                          \
  do {                                                               \
    declspec_stack = tree_cons (build_tree_list (prefix_attributes,  \
                                                 all_prefix_attributes), \
                                current_declspecs,                   \
                                declspec_stack);                     \
  } while (0)
```

다른 하나는 POP_DECLSPEC_STACK macro 이며, 세부 내용은 아래와 같다.

```
#define POP_DECLSPEC_STACK                                                \
  do {                                                                    \
    current_declspecs = TREE_VALUE (declspec_stack);                      \
    prefix_attributes = TREE_PURPOSE (TREE_PURPOSE (declspec_stack));     \
    all_prefix_attributes = TREE_VALUE (TREE_PURPOSE (declspec_stack)); \
    declspec_stack = TREE_CHAIN (declspec_stack);                         \
  } while (0)
```

PUSH_DECLSPEC_STACK macro 의 경우, setspecs 라벨 (Yacc 문법에 정의된) 의 상태에 들어 갈 때마다 호출되게 되며, POP_DECLSPEC_STACK 의 경우, 변수 혹은 함수의 선언이 완료된 후 호출되게 되는 것이 일반적이다.

이제 그럼, 실제로 PUSH_DECLSPEC_STACK 가 호출될 수 있는 상황에 대해 살펴보도록 하자. 나타날 수 있는 상황은 그렇게 다양한 것은 아니다.

- 전역 변수를 선언할 때, 변수의 이름을 처리하기 전에 발생할 수 있을 것이다.
- 함수의 이름을 처리하기 전에 발생할 수 있을 것이다.
- COMPOUND expression 내 앞단에 선언되는 변수를 처리할 때 발생할 수 있을 것이다.

확실하게 말할 수 있는 것은 변수의 선언 혹은 함수의 선언이 이루어질 때, 해당 변수의, 함수의 이름을 처리하기 전에 setspecs 라벨이 호출되어지게 된다는 것이다.

setspecs 라벨이 수행되는 동안, split_specs_attrs () 함수에 의해서, current_declspecs 와 prefix_attributes, all_prefix_attributes 가 업데이트되게 된다. 이렇게 업데이트된 세 개의 전역 변수는 실제로 declarator 라벨을 처리할 때, 사용되기 되며, 대부분의 start_... () 함수가 이러한 정보를 인자로 받아들이게 된다.

제 3 절 Declarator

이 섹션에서는 declarator 에 대해서 살펴보기로 하겠다. 앞 섹션에서 언급한 "Type 의 정의"는 이 declarator 가 가져야 할 type 에 대해 정의를 하는 성격이 짙은데, 이 declarator 에 대해 정의를 한다면, 어떤 변수나 함수의 이름을 나타낸다고 간단하게 언급할 수 있을 것이다.

하지만 declarator 에 대해서 정확하게 언급한다면 아래와 같은 여러 상황을 담는 tree node 라고 할 수 있을 것이다.

- IDENTIFIER
- '*' IDENTIFIER
- IDENTIFIER [expr]
- IDENTIFIER (parmlist)
- (declarator)
- ....

물론 위에서 보인 것에 추가적인 부분까지 여러 상황이 존재할 수 있는데, 결과적으로는 tree node 를 구성하여, start_... () 함수에 전달된다는 것으로 요약할 수 있을 것이다.

그럼 각 상황에 따라 어떠한 declarator tree node 가 생성되게 되는지 살펴보도록 하자.

- int i;

  단순히 전역 변수 형태로 int i 라고 선언하였을 때, declarator 에 해당하는 부분은 "i" 만 해당을 하게 되며, 이에 대한 declarator tree node 는 아래와 같이 생성되게 된다.

  ```
  <identifier_node 0x401819c0 i>
  ```

- int *a;

  이 변수 선언에 대한 declarator 부분은 "*a" 가 해당 부분에 해당되며, 이에 대한 tree node 는 아래와 같이 생성되게 된다.

  ```
  <indirect_ref 0x401713fc
      arg 0 <identifier_node 0x401819c0 a>>
  ```

declarator 가 '*' 로 시작할 경우에는 $prefix/gcc/c-parse.in 파일에 선언되어 있는 make_pointer_declarator () 함수에 의해 생성되는데, 이 함수는 '*' 를 포함하는 absolute declarator 표현을 반환한다. 결과적으로는 INDIRECT_REF 를 반환하게 된다.

- int a[2];

  이 변수 선언에 대한 declarator 부분은 "a[2]" 에 해당하며, 이에 대한 tree node 는 아래와 같이 생성된다.

```
<array_ref 0x4016c5e0
    arg 0 <identifier_node 0x401819c0 a>
    arg 1 <integer_cst 0x4016c5c0
        type <integer_type 0x40166380 int> constant 2>>
```

  ARRAY_REF tree node 의 경우, Yacc 문법의 array_declarator 라벨에서 생성되게 되며, set_array_declarator_type () 함수에 의해, arg 0 에 "a" token 에 해당하는 identifier_node 가 설정되게 되는 것이다.

- int a[];

  위의 예제와 비슷하지만, [...] 내부에 표현식이 존재하지 않을 경우, arg 1 가 설정되지 않은 상태에서 ARRAY_REF tree node 가 생성되게 된다.

```
 <array_ref 0x4016c5c0
    arg 0 <identifier_node 0x401819c0 a>>
```

- int a[2][2];

  만약 이중 배열을 선언하면 어떻게 될지 의문이 들 수 있는데, 여기서 declarator 부분은 "a[2][2]" 부분이라는 사실을 이제 알 수 있을 것이다.

```
<array_ref 0x4016c620
    arg 0 <array_ref 0x4016c5e0
        arg 0 <identifier_node 0x401819c0 a>
        arg 1 <integer_cst 0x4016c5c0 constant 2>>
    arg 1 <integer_cst 0x4016c600
        type <integer_type 0x40166380 int> constant 2>>
```

- int abc ();

  함수의 prototype 선언이 변수의 선언하고 상관없을 거라고 생각하는 분도 계실 수 있겠지만, Yacc 문법에서 이러한 표현식은 fndef 라벨에 존재하는 것이 아닌 datadef 라벨 부분에 해당되는 부분이라고 할 수 있다. 여기서 declarator 부분은 "abc ()" 부분인데, parameter 는 declarator 에 포함된다는 사실을 알기 바란다. 이에 대한 declarator tree node 는 아래와 같이 생성되게 된다.

```
<call_expr 0x4016c5c0
    arg 0 <identifier_node 0x401819c0 abc>
    arg 1 <tree_list 0x401713fc>>
```

  여기서 CALL_EXPR node 의 경우, build_nt () 함수에 의해서 생성되게 된다.

- int abc (int a);

  만약 parameter 가 기입된 prototype 일 경우는, 아래와 같은 tree node 가 생성되게 된다.

```
<call_expr 0x4016c5c0
    arg 0 <identifier_node 0x401819c0 abc>
    arg 1 <tree_list 0x4017149c
        purpose <parm_decl 0x40182540 a type <integer_type 0x40166380 int>
            SI file <stdin> line 1
            size <integer_cst 0x40163540 constant 32>
            unit size <integer_cst 0x401635e0 constant 4>
            align 32 result <integer_type 0x40166380 int>
            initial <integer_type 0x40166380 int>
            arg-type <integer_type 0x40166380 int>
            arg-type-as-written <integer_type 0x40166380 int>>
        chain <tree_list 0x40171474 value <integer_type 0x40166380 int>
            chain <tree_list 0x40171488 value <void_type 0x4016a770 void>>>>>
```

여기서 declarator 부분은 "abc (int a)" 부분이며, parameter 처리가 어떻게 tree node 를 생성되게 되는지는 아래에서 계속 언급해 나갈 것이다.

이제 C 언어상에서 나타날 수 있는 여러 declarator 에 대해서 경우에 따라, 살펴보았는데 이러한 declarator tree node 를 실제로 사용하는 부분에 대해서는 아래의 변수와 함수에서 각각 언급이 되어질 것이다.

제 4 절 표현식 (Expression)

이 섹션에서는 expression 에 대해서 알아 보도록 하자. 일반적인 변수 선언시에는 표현식이 들어가지 않는다. 하지만, 배열을 선언한다고 가장하였을 때, [...] 사이에 표현식이 들어가게 되고 그 표현식은 결과적으로 배열의 크기를 나타낸다. 우선 배열에 대해서 알아보기 이전에, Operator 표현식에 대해서 먼저 알아보도록 하겠다.

4.1 Operator 표현식

앞의 문서를 보면 알겠지만, '+', '−', '*', '/', '%', LSHIFT, RSHIFT, ARITHCOMPARE, EQCOMPARE, '&', '|', '^' 의 경우, 모두 parser_build_binary_op () 함수에 처리한다는 것을 보았다. 그럼 우선 parser_build_binary_op () 함수에서는 어떤 operation 이 일어나는지 살펴보도록 하자. 다른 operator 표현식에 대해서는 계속 아래에서 설명해 나갈 것이다.

이 함수는 binary operator 들을 위해 parser 에 의해 사용되어지는 entry point 인데, 표현식을 만들 뿐만 아니라, 우리는 다른 binary operator 들에 의해 쓰여진 operand 들에 대해 사용자가 혼란스럽게 느낄 수 있는 방식에 대한 검사도 한다. 함수를 살펴보면 알겠지만, 대부분의 수행은 build_binary_op () 에서 이루어진다는 것을 알수 있을 텐데, 그 과정을 살펴보면 아래와 같은 단계를 거친다는 사실을 확인할 수 있을 것이다.

- 인자로 받아들인 arg 0 과 arg 1 의 Type 을 변환하는 부분. 어떠한 operator 를 수행하는데 있어서 가장 중요한 것은 수행할 대상들의 type 이 다를 경우가 발생한다는 것이다. 그리고 그러한 operation 을 정확하게 수행하기 위해서는 어떤 특정 type 으로 변환해 주어야 한다는 것이다. 즉 type 이 narrow 되어야 한다면 좁은, wide 되어야 한다면 넓은 type 을 제대로 설정해 주어야 한다.
- 제대로 설정된 Type 을 통해서, 실제 연산을 할 node 를 만든다. 이 부분에서는 앞 단계에서 처리한 type 과 기존, arg 0 과 arg 1 를 이용하여 실제 AST 수준에서의 연산이 제대로 이루어질 수 있도록 tree node 를 재구성한다.
- EXPR 의 constant folding 과 다른 관련된 simplification 을 수행한다. 관련된 simplification 는 x*1 => x, x*0 => 0, 기타 등등을 포함하고 associative law 의 application 도 포함한다. Operator 표현식 처리의 마지막은 항상 fold () 함수를 호출하게 된다. "Constant folding" 및 "Associative law simplification" 의 경우, 다른 문서에서 언급할 것이다.

아래는 PLUS_EXPR 에 대한 build_binary_op () 함수의 operation 이 요구되었을 때, 함수내에서 수행하는 진행 방법에 대해 간략히 기술한 내용이다.

- PLUS_EXPR

  build_binary_op () 함수의 경우 build () 함수와 다른 점이 있는데, 포인터에 대한 덧셈/뺄셈에 대해서는 special handling 을 하며, 몇몇 최적화가 이루어지게 되는데, PLUS_EXPR 의 경우 다음과 같은 operation 이 일어나게 된다.

  1. Arg 0 과 arg 1 의 각 type 을 default_conversion () 함수를 이용하여 변환을 하게 된다. 이 함수는 표현식에서 사용되는 C data 를 위한 default promotion 들을 수행하며 배열과 함수들은 pointer 로 변환되며, enumeral type 들 혹은 short, chat 는 int 로 변환된다. 게다가, 자신의 값이 분명한 constants symbol 들은 그 값으로 대체된다.
  2. Arg 0 과 arg 1 의 각 type 을 비교를 하는데, constant + constant 형식이 있을 수 있겠지만, pointer + constant 형식이 있을 수 있다. 만약 pointer + constant 형식일 경우 pointer_int_sum () 함수를 통해서 처리하고, constant + constant 일 경우, Arg 0 과 arg 1 의 공통 type 을 계산하도록 한다.
  3. 실제, 공통 type 의 계산은 common_type () 함수에 의해서 수행되며, 계산되어진 공통 type 과 default_conversion () 함수로 변환된 arg 0 과 arg 1 을 인자로 하여, 새로이 PLUS_EXPR node 를 생성한다. 이러한 작업이 필요한 이유는 실제로 PLUS_EXPR 의 arg 0 과 arg 1 이 다른 type 의 값이 될 수 있기 때문이다.
  4. 새로 만들어진 PLUS_EXPR node 를 통해서, "Constant folding"을 수행하게 된다. 그리고 그렇게 만들어진 값을 반환하게 된다.

이제 &a, +1, −1, ++a, −−a, *1, !1, a++, a−− 와 같은 token 을 처리하는 build_unary_op () 함수를 살펴보도록 하자. 이 함수는 unary expression 에 대한 TREE node 건설 및 최적화를 수행하는 함수이다. 하지만 build_binary_op () 함수와 전체적인 수행 맥락은 같은데, 먼저 type 을 정확하게 한 후, 그 type 을 바탕으로 해서, 요청된 node 를 만들게 된다. Assignment 표현식 (a = 1;) 도 마찬가지라고 할 수 있을 것이다. 정확한 Type 설정을 하기 위해서, RHS 의 type 을 LHS 에 의존해 반영하게 되며, convert_for_assignment () 함수를 통해 수행하고, 결과적으로 build () 함수를 통해서, MODIFY_EXPR 을 생성하게 된다.

그 외의 operator 표현식에 대해서는 언급을 하지 않겠는데, 왜냐하면 이 operation 에서 중요한 것은 다음과 같이 요약되기 때문이다.

- Type 의 정확한 변환. convert (), convert_for_assignment () 함수를 사용
- Constant folding 및 Associative law simplification 최적화. AST (Absract Syntax Tree) 의 간단한 최적화.
- 생성 조건이 맞지 않는 오류의 검사.
- 이러한 과정이 완료된 후, 요청된 TREE node 의 생성. build (), build_nt (), build_binary_op (), build_unary_op () 등등의 함수를 통해서 수행한다.

4.2 Statement 표현식

이 하위 섹션은 비록 "표현식" 절에 들어와 있지만, 이 문서의 위에서 부터 아래로 그대로 읽기엔 순서적인 무리가 있기 때문에, "함수" 절을 먼저 읽은 후 이 하위 섹션과 아래 하위 섹션을 계속 읽기 바란다.

그럼 이제부터, statement 표현식에 대해서 알아보도록 하자. 앞 주에서 언급되었던 예제를 바탕으로 해서 적어나가도록 하겠다.

(1) IF 문

IF 문을 살펴보기 이전에, IF 문과 관련된 구조체에 대해서 살펴보도록 하자. if statement 들을 stack 형식으로 관리하기 위해 GCC 에서는 if_stack 이라는 구조체를 사용하는데, if keyword 까지 보인, compound statement 들의 갯수 뿐만 아니라, line number 와 if 가 선언된 file 을 기록하게 되며, 만약 잠재적인 모호한 else 가 보이나면, 그 사실을 또한 기록하게 된다. 그래서 enclosing if statement 가 else branch 를 가지고 있지 않다는 확신이 들 때, 경고를 내보내게 된다. 이에 대한 구조체는 아래와 같다.

```
typedef struct
{
  int compstmt_count;
  int line;
  const char *file;
  int needs_warning;
  tree if_stmt;
} if_elt;
static if_elt *if_stack;
```

이 구조체는 $prefix/gcc/c-common.c 파일에서 살펴볼 수 있고, 이와 더불어 if_stack 구조체를 위한 몇몇 전역 변수에 대해서도 아래에서 살펴보자.

- if_stack_space

  if statement stack 내 공간의 양.

- if_stack_pointer

  Stack pointer

이 구조체 및 전역변수는 c_expand_start_cond () 함수가 수행되면서 설정되게 된다.

또, IF_STMT node 에 쉽게 접근할 수 있도록 accessor 매크로를 GCC 에서 제공하는데, 이에 대해서는 아래와 같다.

```
/* IF_STMT 접근자들. If statement 의 condition 에 대한, if statement 의
   block 에 대한, 만약 else block 이 존재한다면, if statement 의 else block
   에 대한 접근을 도와준다. */
#define IF_COND(NODE)           TREE_OPERAND (IF_STMT_CHECK (NODE), 0)
#define THEN_CLAUSE(NODE)       TREE_OPERAND (IF_STMT_CHECK (NODE), 1)
#define ELSE_CLAUSE(NODE)       TREE_OPERAND (IF_STMT_CHECK (NODE), 2)
```

우선 가장 간단한 if 구문부터 살펴보도록 하자. 대부분의 statement 는 Yacc 문법중 select_or_iter_stmt 라벨에서 시작한다고 앞에서 이야기 하였다. 그중 if 구문은 simple_if 라벨이 ELSE token 앞단까지 처리하고, ELSE token 뒤로는 다시 stmt 라벨이 오도록 배치하여, else if 구문이 있을 수 있는 것을 처리하도록 되어 있다. 그럼 아래와 같은 간단한 if 구문은 어떻게 처리되는지 좀 더 자세히 알아보도록 하자.

```
if (1) { }
```

우선 IF token 을 해석하는 if_prefix 라벨부터 살펴봐야 할 것이다. IF 구문은 condition 을 해석하기 전에 IF_STMT node 를 반드시 build 해야 하는데, 그 이유는 STMT_LINENO 가 "if" 가 있는 줄을 가르키게 하기 위해서 이다. if 구문의 시작은 c_begin_if_stmt () 함수로 시작하는데, 이 함수는 IF_STMT node 를 반환하고, 우리는 나중에 condition 과 다른 부분들을 채울 수 있도록 c_expand_start_cond 에게 전달하게 된다. 우선 c_begin_if_stmt () 함수가 반환한, 처음의 IF_STMT node 를 보면 아래와 같다.

```
<if_stmt 0x401e0600>
```

아주 간단하게 구성되어 있는데, 아직 condition 등의 정보가 기입되지 않았기 때문이다. 이렇게 생성된 IF_STMT node 는 c_expand_start_cond () 함수에 앞에서 읽은 expr 라벨 (IF 구문의 condition. 여기에서의 condition 표현식은 truthvalue_conversion () 함수에 의해서 둘러싸인 모습을 하고 있을 것이다.) 과 함께 전달되게 되는데, 이 함수에서는 if_stack 구조체를 통해서 if statement 에 대한 stack 관리를 하게 된다. 여기에 전달된 condition 의 경우, "1" 이 전부이기 때문에 condition 은 아래와 같은 node 를 보일 것이다.

```
<integer_cst 0x401d7c80 type <integer_type 0x401da380 int> constant 1>
```

이 함수에서 실행되는 구문은 아래의 내용이 전부이다.

```
  IF_COND (if_stmt) = cond;
  add_stmt (if_stmt);

  /* 이 if statement 를 기록한다.  */
  if_stack[if_stack_pointer].compstmt_count = compstmt_count;
  if_stack[if_stack_pointer].file = input_filename;
  if_stack[if_stack_pointer].line = lineno;
  if_stack[if_stack_pointer].needs_warning = 0;
  if_stack[if_stack_pointer].if_stmt = if_stmt;
  if_stack_pointer++;
```

결과적으로 새로이 생성된 IF_STMT node 는 add_stmt () 함수에 의해서 last_tree 에 등록되게 되며, 또한 if_stack 구조체에도 등록되게 된다.

이제 IF 구문의 body 을 해석해야 하는데, 현 예제에서는 선언된 것이 존재하지 않지만 { ... } 가 선언되어 있기 때문에, {, } token 이 어떻게 처리되는지를 보도록 하자. 결과적으로 compstmt_start 라벨과 compstmt_nostart 라벨에서 처리하게 되는데, compstmt_start 라벨에서 COMPOUND_STMT node 가 생성하며, 이 node 를 last_tree 에 등록하여, 위에서 생성된 if_stack 구조체의 CHAIN 으로 등록되게 되며, c_finish_then () 함수에 의해서 IF_STMT node 의 THEN_CLAUSE 에 COMPOUND_STMT 가 등록되게 된다. 그럼 IF_STMT node 의 모습은 최종적으로 아래와 같은 모습일 것이다.

```
<if_stmt 0x4016c600
    arg 0 <integer_cst 0x40163c80 type <integer_type 0x40166380 int> constant 1>
    arg 1 <compound_stmt 0x40171488>>
```

이제 c_finish_then () 함수가 끝마치면, c_expand_end_cond () 함수를 호출하여 If-then 구문의 끝을 기록한다. 이 함수는 전역 변수 if_stack_pointer 를 1 감소시키고, last_expr_type 를 NULL_TREE 로 초기화함으로써 이루어진다.

그럼 IF 구문에 대한 아래와 같은 다른 예제를 보도록 하자.

```
 if (1) { } else { }
```

이 구문은 "if (1) { }" 가 처리될 때까지는 앞의 예제와 같지만 뒤에 { ... } 를 처리하는 부분이 ELSE 부분에 해석하기 때문에, 이에 대한 처리가 추가적으로 이루어지게 된다. ELSE 구문을 처리하기 전에, ELSE 구문의 시작을 알리는 c_expand_start_else () 함수가 호출되며, ELSE 구문의 끝에 도달하면, c_finish_else () 함수와 c_expand_end_cond () 함수가 각각 호출되어 진다. c_expand_start_else () 함수는 then-clause 와 else-clause 사이에 호출되게 되는데, 이 부분에서 enclosing if statement 에 대한 else branch 를 볼 수 없을 경우, ambiguous else 경고를 생성하게 된다. 모호한 else 에 대한 비교하는 조건은 아래와 같다.

```
  if (warn_parentheses
      && if_stack_pointer > 1
      && (if_stack[if_stack_pointer - 1].compstmt_count
          == if_stack[if_stack_pointer - 2].compstmt_count))
    if_stack[if_stack_pointer - 2].needs_warning = 1;
```

이제, c_expand_end_cond () 함수가 실행되고, IF 구문 해석은 마칠 것이다. 그래서 최종적으로 생성된 IF_STMT node 는 아래와 같은 모습을 하게 될 것이다.

```
<if_stmt 0x4016c600
    arg 0 <integer_cst 0x40163c80 type <integer_type 0x40166380 int> constant 1>
    arg 1 <compound_stmt 0x40171488>
    arg 2 <compound_stmt 0x401714b0>>
```

또 다른 if 구문 예제에 대해서 살펴보도록 하자.

```
 if (1) { } else if (1) { }
```

이 예제는 앞의 두 if 구문의 해석 과정이 반복되게 된다. 먼저 예제의 else 까지 위 예제 두번째와 같게 행독하고, 뒤의 if 구문에서는 첫번째 예제와 같은 수행을 하게 된다. 아래에 이를 통해 생성되는 최종 IF_STMT node 가 있다.

```
<if_stmt 0x4016c600
    arg 0 <integer_cst 0x40163c80 type <integer_type 0x40166380 int> constant 1>
    arg 1 <compound_stmt 0x40171488>
    arg 2 <if_stmt 0x4016c620 arg 0 <integer_cst 0x40163c80 1>
        arg 1 <compound_stmt 0x401714c4>>>
```

(2) FOR 문

이제 for 구문에 대해서 알아보도록 하자. FOR_STMT node 를 위한 accessor 들이 존재한다.

```
#define FOR_INIT_STMT(NODE)     TREE_OPERAND (FOR_STMT_CHECK (NODE), 0)
#define FOR_COND(NODE)          TREE_OPERAND (FOR_STMT_CHECK (NODE), 1)
#define FOR_EXPR(NODE)          TREE_OPERAND (FOR_STMT_CHECK (NODE), 2)
#define FOR_BODY(NODE)          TREE_OPERAND (FOR_STMT_CHECK (NODE), 3)
```

for 구문의 경우, node 형성 과정에서 복잡한 형태를 뛰지 않는다. FOR token 을 읽었을 때, 가장 간단한 형태의 FOR_STMT node 를 가지게 되며, 이 때는 초기화, 조건, update 조건이 해석되기 전이니, NULL_TREE 를 가진다. node 생성 후, add_stmt () 를 추가시켜서 last_stmt 를 update 하고, 각각 초기화, 조건, update 조건을 해석한다. "초기화"를 해석한 후 그에 대한 statement 를

```
RECHAIN_STMTS ($<ttype>2, FOR_INIT_STMT ($<ttype>2)); }
```

를 이용하여 update 하게 되며, "조건" 은

```
FOR_COND ($<ttype>2) = truthvalue_conversion ($6);
```

를 통해서 등록되고, 마지막 "update 조건" 또한 xexpr 라벨이 반환한 $$ 값을 인자로 하여

```
FOR_EXPR ($<ttype>2) =$9;
```

에 등록하게 된다. 마지막 BODY 의 경우, last_tree 를 이용하여

```
RECHAIN_STMTS ($<ttype>2, FOR_BODY ($<ttype>2));
```

FOR_BODY 에 넣게 된다. 이 예제에 대한 최종 node 는 아래와 같이 된다.

```
<for_stmt 0x4016c600
    arg 0 <expr_stmt 0x40171474>
    arg 3 <compound_stmt 0x40171488>>
```

(3) DO WHILE 문

DO_WHILE 구문에 대한 accessor 로는 다음과 같은 것들이 존재한다.

```
/* DO_STMT accessor 들.  이것은 각각 do statement 의 condition 과
   do statement 의 body 에 접근할 수 있도록 한다.  */
#define DO_COND(NODE)           TREE_OPERAND (DO_STMT_CHECK (NODE), 0)
#define DO_BODY(NODE)           TREE_OPERAND (DO_STMT_CHECK (NODE), 1)
```

수행 과정은 다음과 같다.

1. DO token 을 해석한 직후, DO_STMT node 를 생성 후 add_stmt () 함수를 이용하여 last_tree 를 update 한다. 그리고 DO_STMT node 의 DO_COND 를 error_mark_node 로 기록한다. 그렇지 않을 경우, 우리는 RTL-generation 때 crash 할 것이다.

2. 이제 body 를 해석한다.
3. WHILE token 을 만났을 때, 앞에서 해석된 body 를 DO_BODY 에 넣는다. 이제 (...) 사이에 있는 condition 을 해석 후, truthvalue_conversion () 함수를 통해 condition 을 변환후, DO_COND 에 넣게 된다.

아래와 같은 예제를 실제 GCC 에서 해석을 했을 경우,

```
do { } while (1);
```

다음과 같은 DO_STMT node 가 생성되게 된다.

```
<do_stmt 0x4016c600
    arg 0 <integer_cst 0x40163c80 type <integer_type 0x40166380 int> constant 1>
    arg 1 <compound_stmt 0x40171474>>
```

(4) WHILE 문

WHILE 구문에 대한 accessor 를 살펴보면 아래와 같다.

```
/* WHILE_STMT accessors.  이것은 각각 while statement 의 condition 과
   while statement 의 body 에 접근할 수 있도록 한다.  */
#define WHILE_COND(NODE)        TREE_OPERAND (WHILE_STMT_CHECK (NODE), 0)
#define WHILE_BODY(NODE)        TREE_OPERAND (WHILE_STMT_CHECK (NODE), 1)
```

WHILE 구문의 경우, c_begin_while_stmt () 함수에 의해서 가장 간단한 WHILE_STMT 가 생성되고, (...) 사이에 존재하는 condition 이 생성되면, c_finish_while_stmt_cond () 함수를 호출하여, WHILE_COND 에 넣게 된다. 물론 당연히 condition 은 truthvalue_conversion () 함수에 의해서 변환이 이루어지게 된다. 여기까지 수행하고 add_stmt () 함수를 통해 last_tree 에 추가한다. 이제 while 의 body 를 읽은 후, WHILE_BODY 에 RECHAIN_STMTS 매크로를 이용하여 body 를 넣은 후 operation 이 마치게 된다. 최종적으로 생성된 while 구문은 아래의 예제를 사용하였을 때, 다음과 같게 된다.

```
while (1) { }
```

에 대해 다음의 결과가 나온다.

```
<while_stmt 0x4016c600
    arg 0 <integer_cst 0x40163c80 type <integer_type 0x40166380 int> constant 1>
    arg 1 <compound_stmt 0x40171488>>
```

(4) SWITCH 문

SWITCH 구문에 대한 accessor 로는 아래의 것이 존재한다.

```
/* SWITCH_STMT accessors.  이것은 각각 switch statement 의 condition 과
   body, original condition (어떤 compiler 변환이 있기 전) 에 대해 접근하도록
   한다.  */
#define SWITCH_COND(NODE)       TREE_OPERAND (SWITCH_STMT_CHECK (NODE), 0)
#define SWITCH_BODY(NODE)       TREE_OPERAND (SWITCH_STMT_CHECK (NODE), 1)
#define SWITCH_TYPE(NODE)       TREE_OPERAND (SWITCH_STMT_CHECK (NODE), 2)
```

SWITCH 구문의 경우, struct c_switch 구조체를 stack 형식으로 사용되는데, 이에 대해서 잠시 살펴보도록 하자.

```
struct c_switch {
  tree switch_stmt;
  splay_tree cases;
  struct c_switch *next;
};
```

각 element 는 아래와 같은 내용이다.

- switch_stmt

  생성중인 SWITCH_STMT.

- cases

  case range 의 low element 를 high element 로 mapping 하는 splay-tree 새 case label 이 이전 case label 과 중복되는지 아닌지를 결정하는데, 사용된다. 우리는 간단한 hash table 보다, tree 가 필요한데, 왜냐하면 GNU case range extension[1] 때문이다.

- next

  Stack 상의 다음 node.

또한 현재 active switch statement 가르키지 위한 전역 변수가 하나 존재하는데, 아래와 같다.

```
/* 현재 active switch statement 들의 stack.  가장 내부 switch statement
   는 stack 의 top 에 있다.  함수의 body 를 처리하는 동안 이 stack 이
   유일하게 active 하기 때문에 garbage collection 에 stack 을 기록할 필요가
   없다.  또한 우리는 이 시점에서는 전혀 collect 를 하지 않는다.  */
static struct c_switch *switch_stack;
```

SWITCH 구문의 경우 (...) 내에 포함된 표현식은 integer 여야 하는데, SWITCH token 을 해석하고 (...) 내부를 해석한 후, c_start_case () 함수에 의해서 condition 에 대한 검사를 실시한 후 SWITCH_STMT node 가 새로이 생성되게 된다. 또한 struct c_switch 를 update 하게 된다.

```
  /* Stack 에 새 SWITCH_STMT 를 더한다.  */
  cs = (struct c_switch *) xmalloc (sizeof (*cs));
  cs->switch_stmt = build_stmt (SWITCH_STMT, exp, NULL_TREE, orig_type);
  cs->cases = splay_tree_new (case_compare, NULL, NULL);
  cs->next = switch_stack;
  switch_stack = cs;
```

그런후 add_stmt () 를 통해 last_tree 에 등록을 하게 된다.
아래의 예로 설명을 하도록 하면 다음과 같게 된다.

```
switch (1) { case 'a': break; default: break;}
```

1. switch (1) 부분까지가 위에서 설명한 switch_stack 에 SWITCH_STMT 는 구성요소로써 넣는 부분이다.

2. 이제 각 라벨의 경우, Yacc 문법 중 case 구문을 다루는 부분은 label 라벨에서 이다. 실제 각 case 구문은 do_case () 함수를 호출하지만 실제 처리되는 부분은 이 함수내에서 호출되는 c_add_case_label () 함수이다. 이 예제의 경우, GNU case extension 이 없기 때문에, 이에 대해서는 신경쓰지 않겠다. 실제 CASE 구문의 'a' 는 c_add_case_label () 함수의 low_value 로 들어가게 되고, check_case_value () 함수를 통해서, 값을 검사한 후, convert_and_check () 함수를 통해서, SWITCH 구문에 선언된 condition 이 가지는 type 으로 변환하게 된다. 실제로 case 구문의 경우, splay_tree 를 이용하여 관리하게 되는데, 중복되는 case 표현식이 없는지 확인하게 된다. GNU case extension 의 경우 이 case range 와 다른 case range 가이에 overlap 이 있을 수 있고 우리는 어떠한 overlapping case range 들을 허락하지 않기 때문에, LOW_VALUE 보다 작은 가장 큰 low case 를 찾고, LOW_VALUE 보다 큰 가장 작은 case label 을 찾을 필요가 있다. 만약 overlap 이 존재한다면, 두 range 들의 하나에서 발생할 것이다.

[1] 보통 case 구문의 경우 범위를 지정해 줄 수 없는데 대부분이지만, GNU 컴파일러에서는 'a' ... 'b' 와 같이 선언하여 CASE 구문의 범위를 지정해 줄 수 있다.

```
<case_label 0x4016c660
    arg 0 <integer_cst 0x4016c640 type <integer_type 0x40166380 int> constant 97>
    arg 2 <label_decl 0x40182700 VOID file <stdin> line 2
        align 1 context <function_decl 0x401825b0 main>>>
```

이제 statement-tree 에 CASE_LABEL 를 더하게 되며, splay_tree 에 또한 low_value 를 key 로 하고 생성된 CASE_LABEL 을 값으로 하여 등록되게 된다. 위에 보이는 CASE_LABEL 이 case 'a' 에 해당하는 node 인 것이다.

3. 이제 break 구문이 해석되어 BREAK_STMT node 가 생성되어 statement-tree 에 더하게 된다. statement-tree 더한다는 것이 add_stmt () 함수를 통해 last_tree 를 update 한다는 것이라고 앞에서 언급하였다.

결과적으로 아래와 같은 SWITCH node 가 만들어지게 된다.

```
<switch_stmt 0x4016c600
    arg 0 <integer_cst 0x40163c80 type <integer_type 0x40166380 int> constant 1>
    arg 1 <compound_stmt 0x40171488
        arg 0 <scope_stmt 0x4017149c tree_0
            chain <case_label 0x4016c660
                arg 0 <integer_cst 0x4016c640 constant 97>
                arg 2 <label_decl 0x40182700>
                chain <break_stmt 0x40176a00
                    chain <case_label 0x4016c680
                        arg 2 <label_decl 0x40182770>
                        chain <break_stmt 0x40176a10
                            chain <scope_stmt 0x401714d8>>>>>>>
    arg 2 <integer_type 0x40166380 int SI
        size <integer_cst 0x40163540 constant 32>
        unit size <integer_cst 0x401635e0 constant 4>
        align 32 symtab 0 alias set -1 precision 32
        min <integer_cst 0x401635a0 -2147483648>
        max <integer_cst 0x401635c0 2147483647>
        pointer_to_this <pointer_type 0x4016e620>>>
```

위의 node 를 보면, 중간에 SCOPE_STMT 가 들어간 부분에 대해서 설명을 하도록 하겠다. 이 부분은 compstmt_nostart 라벨의 하위 라벨들인 pushlevel 라벨 (pushlevel () 함수 혼동하지 마세요.) 에서 이루어지는 operation 이다. pushlevel 라벨에서는 { ... } 와 같은 새로운 sope 가 발생할 경우, Binding level 을 pushlevel () 함수를 통해서 만들고, last_expr_type 를 0 으로 설정하고 add_scope_stmt () 함수를 통해서 SCOPE_STMT 를 statement list 에 추가하게 된다. add_scope_stmt () 함수의 설명은 다음과 같다.

> statement-tree 에 scope-statement 를 추가한다. BEGIN_P 는 이 statements 가 scope 를 open 혹은 close 했는지 가르킨다. PARTIAL_P 는 부분적인 scope 에 대해서 true 인데, 즉 scope 는 cleanup 이 필요한 object 가 생성되었을 때 라벨 뒤부터 시작한다. 만약 BEGIN_P 가 0 이 아니라면 SCOPE_STMT stack 의 top 를 나타내는 새 TREE_LIST 를 반환한다. TREE_PURPOSE 는 새 SCOPE_STMT 이다. 만약 BEGIN_P 가 0 이면 TREE_VALUE 가 새 SCOPE_STMT 이고 TREE_PURPOSE 가 SCOPE_BEGIN_P set 과 매칭하는 SCOPE_STMT 인 TREE_LIST 를 반환한다.

결과적으로 SWITCH 구문의 condition 이 parsing 된후, CASE 구문까지 모두 해석하고 마지막 poplevel 라벨을 통해 add_scope_stmt () 함수가 실행될 때는 아래와 같은 모습을 전역 변수 c_scope_stmt_stack 가 가지게 된다. c_scope_stmt_stack 변수는 현재의 scope statement stack 을 나타낸다.

```
<tree_list 0x401714b0
    purpose <scope_stmt 0x4017149c tree_0
        chain <case_label 0x4016c660
            arg 0 <integer_cst 0x4016c640 constant 97>
            arg 2 <label_decl 0x40182700>
            chain <break_stmt 0x40176a00
                chain <case_label 0x4016c680 arg 2 <label_decl 0x40182770>
                    chain <break_stmt 0x40176a10>>>>>
    value <scope_stmt 0x401714d8>
    chain <tree_list 0x40171460
        purpose <scope_stmt 0x4017144c tree_0
            chain <switch_stmt 0x4016c600
                arg 0 <integer_cst 0x40163c80 constant 1>
                arg 2 <integer_type 0x40166380 int>
                chain <compound_stmt 0x40171488
                    chain <scope_stmt 0x4017149c>>>>>>
```

위의 TREE_LIST node 의 생성과정에 대해서 짧게 설명하면, SWITCH 구문의 condition 이 해석된 후 add_scope_stmt () 함수가 호출되는 과정에서 TREE_LIST 의 CHAIN 에 기존의 c_scope_stmt_stack 값이 들어가게 되고, purpose 는 단순한 형태의 SCOPE_STMT 만 가졌을 것이다. 즉, SCOPE_STMT node 의 chain 은 없었을 것이다. 하지만, 구문이 해석되면서 add_stmt 에 의해서 이에 대한 node 들이 chain 으로 묶이게 되어 TREE_LIST 의 chain 과 purpose 가 위의 모양을 가지게 되었을 것이다. 아직 value 는 설정되지 않았다. 그럼 SWITCH 구문의 마지막에 poplevel 라벨에 의해서 add_scope_stmt () 함수가 다른 인자 값을 가진체 호출되게 되는데, 이 부분에서 TREE_LIST 의 value 가 scope_stmt 를 가지게 된다.

SCOPE_STMT 는 몇 개의 accessor 가 존재하는데, 이에 대해서 마지막으로 알아보도록 하자.

```
/* 만약 이 SCOPE_STMT 가 scope 의 시작을 위한 것이라면 0 이 아닌 값을 가진다.  */
#define SCOPE_BEGIN_P(NODE) \
  (TREE_LANG_FLAG_0 (SCOPE_STMT_CHECK (NODE)))
/* 만약 이 statement 가 partial scope 를 위한 SCOPE_STMT 일 경우, 0 이
   아닌 값을 가진다.  예를 들면 아래와 같다.

     S s;
     l:
     S s2;
     goto l;

   'l' 뒤에는 비록 curly brace ( {...} 같은) 가 없지만, (묵시적으로) 새
   scope 가 있다.  특히 우리가 goto 를 hit 할때 우리는 s2 를 반드시 destroy
   를 하고 그것을 재 construct 해야 한다.  묵시적인 scope 를 위해,
   SCOPE_PARTIAL_P 가 설정될 것이다.  */
#define SCOPE_PARTIAL_P(NODE) \
  (TREE_LANG_FLAG_4 (SCOPE_STMT_CHECK (NODE)))
```

4.3 Label 표현식

이제 label 표현식에 대해서 알아보도록 하자. 우선 label 표현식에 대한 적당한 예를 들도록 하자.

```
int
main ()
  int i = 0;
l:
  i++;
  goto l;
```

위의 예제가 어떻게 해석되는지에 대해서 알아보면, 실제 라벨 L 이 해석되는 부분은 Yacc 문법의 label 라벨에서 해석되며, 이러한 패턴을 만났을 때, GCC 에서 수행하는 것은 우선, define_label () 함수를 통해서, LABEL_DECL node 를 생성하게 되는데, 이렇게 생성된 node 에 decl_attributes () 함수를 통해서 label 에 주어진 attribute 를 node 에 적용하고, LABEL_STMT node 를 LABEL_DECL node 를 인자로 하여 생성할 것이다. 그런 후 Statement Tree 에 더한다. 실제 생성된 node 를 살펴보면 아래와 같은 모양을 가지고 있다.

```
<label_stmt 0x40171488
    arg 0 <label_decl 0x40182700 l
        type <void_type 0x4016a770 void VOID
            align 8 symtab 0 alias set -1
            pointer_to_this <pointer_type 0x4016a7e0>>
        VOID file <stdin> line 2
        align 1 context <function_decl 0x401825b0 main>
        initial <error_mark 0x40168a20>>>
```

이제 GOTO 문을 보도록 하자. GOTO 문의 정의는 Yacc 문법의 stmt 라벨에 정의되어 있으며, lookup_label () 함수를 통해서, 앞에서 정의한 LABEL_DECL node 를 identifier 를 기반으로 가지게 된다. 즉 위에서 LABEL_DECL node 가 생성될 때, 해당 identifier 의 IDENTIFIER_LABEL_VALUE 가 이 LABEL_DECL node 로 등록되게 된다. 이렇게 찾은 node 를 이용하여, GOTO_STMT node 를 생성한 후, Statement Tree 에 더한다. 선언된 node 는 아래와 같은 모양을 가지게 될 것이다.

```
<goto_stmt 0x401714d8
    arg 0 <label_decl 0x40182700 l
        type <void_type 0x4016a770 void VOID
            align 8 symtab 0 alias set -1
            pointer_to_this <pointer_type 0x4016a7e0>>
        used VOID file <stdin> line 2
        align 1 context <function_decl 0x401825b0 main>
        initial <error_mark 0x40168a20>>>
```

제 5 절 변수

C 언어에서의 변수는 많은 곳에서 나타날 수 있고, 선언되어질 수 있는데, 전역 변수로 지정될 수도, 함수를 선언할 때의 parameter 로도, 함수내부에서도 선언되어질 수 있다. 이 섹션에서는 변수를 지정할 때, 어떠한 operation 이 GCC 내에서 발생되는지 살펴보도록 하자.

22 주 문서 "C 언어를 위한 Yacc 문법" 에서 이미 살펴보았듯이, 변수의 경우 각 변수에 대한 초기값이 존재하느냐 하지 않느냐에 따라, 약간의 다른 수행이 이루어지게 되며, 초기값이 존재하지 않을 경우, start_decl () 함수와 finish_decl () 함수만 호출되지만 초기화값이 있을 경우, start_init () 함수와 finish_init () 함수가 또한 호출된게 된다.

5.1 초기화가 없는 변수 정의 (start_decl)

만약 해당 변수에 대한 초기화값이 존재하지 않을 경우에 대해서 먼저 살펴보도록 하자. 앞에서 언급했듯이, 이러한 경우 start_decl () 함수와 finish_decl () 함수가 호출되게 되는데, 초기화가 없는 변수의 선언은 여러 가지 유형이 있을 수 있는데, 하나 하나 나열할 수는 없으므로 크게 나누어 생각해 보겠다. (1) 우선 변수의 선언에서 가장 앞단에 올 수 있는 부분이다.

- SC (storage class)
- TS (type specifier)
- SA (start attribute)

- EA (end attribute)

(2) 그리고 실제 declarator 의 이름을 지정해 줄 수 있는 부분이 바로 뒤에 나오게 되는데, 다음과 같은 여러 상황이 있을 수 있다.

- IDENTIFIER
- '*' IDENTIFIER
- IDENTIFIER [ ]
- IDENTIFIER (parmlist)
- (declarator)

(3) 그리고 다시 이 부분 뒤에는

- ASM
- ATTRIBUTE 들

이 올 수 있다. 위에서 언급한 크게 세 부분이 초기화가 정의되어 있지 않은 변수를 구성하는 부분이다.

실제로 start_decl () 함수와 finish_decl () 함수가 호출되는 시기는 (3) 단계까지 수행이 완료된 후에 호출이 되는데, 여기서 주의해야 할 부분은 (1) 단계가 수행된 후에는 그에 대한 tree node 가 전역 변수 current_declspecs 와 all_prefix_attributes 에 이미 들어가 있기 때문에, (2) 단계, (3) 단계가 반복되며 수행이 되며, (1) 단계는 한차례만 수행이 이루어진 후 다시 수행되지 않는다는 점이다. 그래서 아래와 같이 같은 type 에 여러개의 변수를 선언해야 할 때가 있을 경우, 이를 통해 해석을 원할하게 할 수 있게 된다.

```
int a, b, c;
```

실제로, start_decl () 함수가 호출되어질 때, 여러가지 parameter 들이 전달이 되는데, 전역변수 all_prefix_attributes 에 들어 있는 tree node 와 (3) 단계에서 해석된 ATTRIBUTE 들은 chainon () 함수에 의해 연결되어 전달된다는 것을 알기 바란다. start_decl () 함수에 전달되는 내용은 다음과 같다고 할 수 있겠다.

- Declarator

  이 tree node 는 앞의 (2) 단계에서 해석된 tree node 를 가지고 있다.

- Declarator specs

  이 tree node 는 전역 변수 current_declspecs 에 저장되어 있는 node 정보이다.

- 초기화 여부

  만약 parameter 인 initialized 변수의 값이 0 일 경우, 초기화를 수행하지 않는다는 것이고, 1 일 경우 초기화값을 가지고 있다는 것이다.

- Attribute 들

  앞에서 이 변수를 실제로 생성하기까지 모은 ATTRIBUTE tree node 정보를 가지고 있는 전역 변수 all_prefix_attributes 에 (3) 단계에서 해석된 ATTRIBUTE 들은 chainon () 한 node 가 전달되게 된다.

그럼 이제부터 start_decl () 함수의 역활에 대해서 알아보고, 실제로 처리 세부 과정에 대해서 살펴보도록 하자.

start_decl () 함수는 보통의 선언 (declaration) 혹은 data 정의 (definition) 에서의 선언자 (declarator) 를 해독하는 역활을 하는데, 이 함수는 만약 초기화자 (initializer) 가 존재한다면 그전에 type 정보와 변수 이름을 해석하자마자 호출되어진다. 여기서 우리는 ...._DECL node 를 생성하며 그것의 type 부분을 채우고 그것을 현재 context 를 위한 decl 들의 리스트에 그것을 올려놓는 역활을 한다. ...._DECL node 는 값으로써 반환되어진다. 처리과정에서 예외사항이 존재를 하는데, 길이가 정의되지 않는 배열의 경우 type 은 NULL 로 남겨지게 되고 그것은 이후에 호출되는 'finish_decl () 함수' 에서 채워지게 된다.

start_decl () 함수는 다음과 같이 크게 나누어 생각 되어질 수 있다.

1. 주어진 declarator, current_declspecs, 초기화여부 정보를 이용하여 그에 맞는 ..._DECL tree node 를 구성하는 부분
2. 초기화값이 현재 선언하는 변수에게 존재하여도 되는 것인지를 확인하는 과정 및 DECL_INITIAL (decl) 설정하는 부분. 이 설정을 통해, 'pushdecl' 함수에서 이것은 초기화된 decl 임을 말해주고, 'finish_decl' 함수에게 실제 초기화자를 저장할 것이라는 것을 간접적으로 말한다.
3. ..._DECL tree node 에 위에서 parameter 로 건네받은 attribute 들을 설정하는 부분
4. 현재 binding level 에 이 ..._DECL tree node 를 추가하는 부분.

이제 각 부분에 대해서 좀 더 세부적으로 살펴보도록 하자.

5.1.1 ..._DECL tree node 를 구성하는 부분

start_decl () 함수가 호출될 때 건네진, declarator, declspecs, initialized 변수를 이용하여, grokdeclarator () 함수를 호출하게 된다. 이 함수의 역활은 주어진 declspecs 와 declarator 를 통해서 선언된 object 의 이름과 type 을 결정하고 그것을 위한 ..._DECL node 를 건설하게 된다. (어떤 경우 우리는 .._TYPE 을 대신 반환할 수 있으며, 잘못된 입력의 경우 때때로 0 을 반환한다.) 인자로 넘어가게 되는 각 argument 에 대한 것을 살펴보면 아래와 같이 설명을 할 수 있는데,

1. DECLSPECS 는 필드의 값이 storage class 들과 type specifier 들로 구성된 tree_list node 들의 chain 이다.
2. DECLARATOR 는 앞에서 설명하였다.
3. DECL_CONTEXT 는 이 declaration 이 어떤 syntactic context 에 속하는지 말한다

   NORMAL : 대부분의 context 용. VAR_DECL 혹은 FUNCTION_DECL 혹은 TYPE_DECL 를 만듭니다.

   FUNCDEF : 함수 정의 용. NORMAL 하고 비슷하지만 각각의 경우에 따라 오류 메세지에서 약간의 차이가 있습니다. 만약 이 정의가 해석하기에는 너무 터무니없을 경우, 이를 전달하기 위해 0 을 반환할 수 있다.

   PARM : parameter 선언 용. (함수 prototype 내에 내장된 것이거나 함수 몸체(body) 앞에 있는 것) PARM_DECL 을 만들거나 void_type_node 를 반환합니다.

   TYPENAME 는 typename (cast 이거나, sizeof 인) 에 대한 것이다. DECL node 를 만들지 않습니다; 단지 ..._TYPE node 를 반환합니다.

   FIELD 는 struct 혹은 union 필드를 위한 것; FIELD_DECL 를 만듭니다.

   BITFIELD 는 지정된 길이를 가지는 필드.
4. INITIALIZED 는 만약 decl 이 초기화자(initializer)를 가지고 있다면 1 을 가집니다.

결과적으로 grokdeclarator () 함수가 해석되게 되는 tree node 의 CODE 는 ARRAY_REF, INDIRECT_REF, CALL_EXPR, TREE_VALUE, IDENTIFIER_NODE 로 압축될 수 있다. 이를 벗어난 CODE 에 대해, grokdeclarator () 함수를 호출해서는 안된다. grokdeclarator () 함수는 결과적으로 ..._DECL node 를 반환하는 것이 목적으로 반환되는 node 를 살펴볼 경우, 아래와 같은 node 가 반환되어질 수 있음을 확인할 수 있다.

- TYPE_DECL
- PARM_DECL
- FIELD_DECL
- FUNCTION_DECL

- VAR_DECL

...DECL node 를 생성할 때는, build_decl () 함수에 의해서 생성되게 된다. 내부에서 사용되는 특별한 의미를 가지는 구조체라던가, 전역 변수는 존재하지 않지만, 결과적으로 ...DECL node 를 생성하는데 있어서 영향을 미치는 것은 당연하다. 그럼 실제로 ...DECL node 를 생성하는데 영향을 미치는 변수들에 대해서 살펴보고, 이러한 것이 어떻게 설정되는지 알아보도록 하자.

- **decl_context** 는 앞에서 설명하였듯이, 해당 변수가 선언된 syntactic context 가 어디에 속하는지를 말해준다. 어떤 변수를 예로 든다면, C 언어에서 전역 변수로 선언될 수도 있고, 어떤 parameter 의 인자로 선언될 수도 있고, 로컬 변수로 선언될 수 도 있다. 그에 대한 위치를 상대적으로 가르켜 주기 위해서 존재한다.

- **specbits** 의 경우, enum rid 에 대한 mask 값을 가지는 변수이다. 자세한 사항은 enum rid 의 주석을 살펴보면 알 수 있는데, 아래와 같이 설명이 되어 있다.

  > [enum rid 설명] 예약된 식별자들. 이것은 C 와, C++, Objective C 에서 사용되는 모든 key 값들의 조합입니다. 모든 type modifier 들은 시작시 하나의 block 내에 반드시 있어야 하는데 그것은 mask bit 로써 사용되기 때문이다. 27 개의 type modifier 들이 있습니다; 만약 우리가 좀 더 추가시킨다면 mask mechanism 을 반드시 재설계 해야 할 것입니다.

  이 값의 상태에 따라, 내부 변수의 **constp**, **restrictp**, **volatilep**, **inlinep** 값이 0 혹은 1 로 설정이 되며, 이 네가지 값에 따라, type_quals 값 또한 변하게 된다. 이 변수는 지정된(specified) modifier 들을 처리하고 잘못된 조합(combination)들을 검사하는데 사용되게 된다. 예를 든다면, 두가지 storage class 들이 주어졌을 경우가 있을 수도 있으며, 같은 storage class 가 두번 이상 선언되었을 경우에도 문제가 될 수 있기 때문에, 이에 대한 경고 메세지를 생성할 수 있다.

- **typedef_decl**, **typedef_type** 의 경우, 현재 해석하고 있는 declspecs 에서의 정보를 가지게 되는데, typedef_decl 변수의 경우, Type 에 대한 tree node 인 TYPE_DECL node 을 가지게 되며, typedef_type 의 경우, 이 TYPE_DECL node 의 type 정보를 가지게 된다. 'int' 와 같이 미리 생성되어 있는 node 의 경우에는 lookup_name () 함수를 통해 해당 node 를 찾아 이에 대한 정보를 기록하게 된다.

- **innermost_code** 는 함수 정의 (definition) 의 declarator 가 함수 선언자 형식을 가지고 있는지 검사하는데 사용된다.

- 기타 다른 변수도 존재하나, 수행 결과에 영향을 크게 미치지 않는 것은 제외하였다.

그럼, 각 상황에 따라서, ...DECL node 가 어떻게 구성되는지, 직접 tree debugging 을 통해서 살펴보도록 하자.

- int i; (전역 변수로 선언될 경우)

  변수 i 의 경우, 전역 변수로 선언되기 때문에, 지역 변수와 약간 차이가 있다고 할 수 있는데, grokdeclarator () 함수로 전달된 argument 들의 내용은 아래와 같다고 할 수 있다.

  - declarator

    ```
    <identifier_node 0x401819c0 i>
    ```

  - declspecs

    ```
    <tree_list 0x401713c0 static
        value <identifier_node 0x40169100 int tree_0
            global <type_decl 0x40166a80 int
                type <integer_type 0x40166380 int>
                VOID file <built-in> line 0
                align 1>
            rid 0x40169100 "int">>
    ```

grokdeclarator () 함수에서 발생하는 세부 자세한 operation 에 대해서는 직접 코드를 보길 바라며, 여기에서는 간략하게 지역 변수에 어떻게 설정되는지만 볼 생각이다. 지역 변수 name 에 "i" 가 들어가게 되며, 지역 변수 explicit_int 는 1, typedef_decl 는 "int" 에 대한 TYPE_DECL node 가 typedef_type 에는 typedef_decl 의 TREE_TYPE 이 들어가게 된다. 위와 같이 선언된 경우, 지역 변수 specbits 는 0 이 설정되어 constp, restrictp, volatilep, inlinep 의 값은 모두 0 으로 설정되어 결과적으로 다음과 같은 VAR_DECL 이 생성되게 되는 것이다.

```
<var_decl 0x40182540 i
    type <integer_type 0x40166380 int SI
        size <integer_cst 0x40163540 constant 32>
        unit size <integer_cst 0x401635e0 constant 4>
        align 32 symtab 0 alias set -1 precision 32
        min <integer_cst 0x401635a0 -2147483648>
        max <integer_cst 0x401635c0 2147483647>
        pointer_to_this <pointer_type 0x4016e620>>
    SI file <stdin> line 1 size <integer_cst 0x40163540 32>
    unit size <integer_cst 0x401635e0 4>
    align 32>
```

이와 같은 VAR_DECL 이 생성된 후, 이 변수가 전역변수인지 로컬변수인지에 따라서 최종적으로 생성되는 ...._DECL 의 구성 요소 TREE_PUBLIC (decl) 와 TREE_STATIC (decl) 의 값이 달라지게 된다. 최종적으로 생성된 VAR_DECL 은 아래와 같게 된다.

```
<var_decl 0x40182540 i
    type <integer_type 0x40166380 int SI
        size <integer_cst 0x40163540 constant 32>
        unit size <integer_cst 0x401635e0 constant 4>
        align 32 symtab 0 alias set -1 precision 32
        min <integer_cst 0x401635a0 -2147483648>
        max <integer_cst 0x401635c0 2147483647>
        pointer_to_this <pointer_type 0x4016e620>>
    public static SI file <stdin> line 1 size <integer_cst 0x40163540 32>
    unit size <integer_cst 0x401635e0 4>
    align 32>
```

- char a; (로컬 변수로 선언될 경우)

  위 예제와 거의 같은 예제이며, 함수내에서 선언된 변수에 대한 것이다.

  - declarator

    ```
    <identifier_node 0x401819c0 i>
    ```

  - declspecs

    ```
    <tree_list 0x40171474 static
        value <identifier_node 0x40165d80 char tree_0
            global <type_decl 0x40166af0 char
                type <integer_type 0x40166230 char>
                VOID file <built-in> line 0
                align 1 chain <type_decl 0x40166a80 int>>
            rid 0x40165d80 "char">>
    ```

  지역변수 explicit_char 가 1 로 설정. 결과적으로 나오는 VAR_DECL 은 아래와 같다. 그 외의 모든 부분이 전역 변수 int a; 선언과 같음.

  ```
  <var_decl 0x40182700 a
      type <integer_type 0x40166230 char QI
  ```

```
        size <integer_cst 0x401633e0 constant 8>
        unit size <integer_cst 0x40163400 constant 1>
        align 8 symtab 0 alias set -1 precision 8
        min <integer_cst 0x401634a0 -128> max <integer_cst 0x401634c0 127>
        pointer_to_this <pointer_type 0x4016ac40>>
    QI file <stdin> line 2 size <integer_cst 0x401633e0 8>
    unit size <integer_cst 0x40163400 1>
    align 8>
```

- extern int i;

  전역 변수 형태로 선언된 것이며, 앞에 extern token 이 붙어있다.

  – declspecs

```
    <tree_list 0x401713d4 static
        value <identifier_node 0x40169100 int tree_0
            global <type_decl 0x40166a80 int
                type <integer_type 0x40166380 int>
                VOID file <built-in> line 0
                align 1>
            rid 0x40169100 "int">
        chain <tree_list 0x401713c0
            value <identifier_node 0x40165f80 extern tree_0
                rid 0x40165f80 "extern">>>
```

  – declarator

```
    <identifier_node 0x401819c0 i>
```

  extern 이 정의되어 있기 때문에, 지역변수 specbits 의 값이 0x10 (2 진수로 10000) 으로 설정됨. 이것은 enum rid 의 RID_EXTERN 의 mask 를 의미. 이로 인해 extern_ref 가 1 로 설정되게 된다. 그래서 DECL_EXTERNAL (decl) 에 정보가 기록되게 되며, 이 예제는 전역변수 상태에서 선언된 것이기 때문에 아래와 같이 VAR_DECL 이 만들어 지게 된다.

```
<var_decl 0x40182540 i
    type <integer_type 0x40166380 int SI
        size <integer_cst 0x40163540 constant 32>
        unit size <integer_cst 0x401635e0 constant 4>
        align 32 symtab 0 alias set -1 precision 32
        min <integer_cst 0x401635a0 -2147483648>
        max <integer_cst 0x401635c0 2147483647>
        pointer_to_this <pointer_type 0x4016e620>>
    public external SI file <stdin> line 1
    size <integer_cst 0x40163540 32> unit size <integer_cst 0x401635e0 4>
```

- int *a;

  – declspecs

```
    <tree_list 0x401713c0 static
        value <identifier_node 0x40169100 int tree_0
            global <type_decl 0x40166a80 int
                type <integer_type 0x40166380 int>
                VOID file <built-in> line 0
                align 1>
            rid 0x40169100 "int">>
```

  – declarator

```
<indirect_ref 0x401713fc
    arg 0 <identifier_node 0x401819c0 a>>
```

declarator 를 해석하는 동안, 지역변수 innermost_code 가 설정된다. declarator 가 단순한 IDENTIFIER 가 아닌 조금 더 복잡한 구조로 이루어질 경우, node 를 감소시켜 가면서, 좀 더 복잡한 type 을 우리가 선언된 identifier (혹은 absolute declarator 내 존재하는 NULL_TREE) 에 도달될 때까지 생성하게 된다. declarator 가 INDIRECT_REF node 이기 때문에, declspecs 에서 구한 "int" 의 TREE_TYPE 을 build_pointer_type () 함수를 이용하여 POINTER_TYPE node 를 생성하게 된다. 결과적으로 아래와 같은 POINTER_TYPE 이 반환되게 된다.

```
<pointer_type 0x4016e620
    type <integer_type 0x40166380 int SI
        size <integer_cst 0x40163540 constant 32>
        unit size <integer_cst 0x401635e0 constant 4>
        align 32 symtab 0 alias set -1 precision 32
        min <integer_cst 0x401635a0 -2147483648>
        max <integer_cst 0x401635c0 2147483647>
        pointer_to_this <pointer_type 0x4016e620>>
    unsigned SI
    size <integer_cst 0x40163b80
        type <integer_type 0x4016a540 bit_size_type> constant 32>
    unit size <integer_cst 0x40163be0
        type <integer_type 0x4016a4d0 unsigned int> constant 4>
    align 32 symtab 0 alias set -1>
```

이게 '*' 로 인해서 POINTER_TYPE 을 만들었기 때문에, 이제 declarator 로 arg 0 인 IDENTIFIER_NODE 로 바꾸게 되며, 위의 다른 예제와 마찬가지로 POINTER_TYPE 와 IDENTIFIER_NODE 로 바뀐 declarator 를 사용하여, VAR_DECL node 를 마지막으로 생성한다.

- int a[2];
  - declspecs

```
<tree_list 0x401713c0 static
    value <identifier_node 0x40169100 int tree_0
        global <type_decl 0x40166a80 int
            type <integer_type 0x40166380 int>
            VOID file <built-in> line 0
            align 1>
        rid 0x40169100 "int">>
```

  - declarator

```
<array_ref 0x4016c5e0
    arg 0 <identifier_node 0x401819c0 a>
    arg 1 <integer_cst 0x4016c5c0
        type <integer_type 0x40166380 int> constant 2>>
```

초기에 innermost_code 가 설정되는 부분은 위 예제와 같으며, 이 예제는 배열이기 때문에, ARRAY_REF 의 처리가 이루어질 것이며, 실제로 배열의 크기는 2 라고 정의되어 있지만, 0 부터 1 까지의 총 2 개의 공간을 할당하게 된다는 의미를 나타낼 것이다. 이러한 것을 처리하기 위해서는 arg 0 에 있는 것은 declarator 로 재 지정하고, arg 1 에 있는 것을 재처리하게 되는데, arg 1 은 배열의 실제 크기를 지정하는 표현식이다. 비록 이 예제에서는 2 와 같이 단순히 constant 를 넣었지만, 복잡한 표현식이 이곳에 나타날 수 있을 것이다. 결과적으로는 GCC 는 ARRAY_TYPE node 를 생성하여야 하는데, 고려해야 할 것이 있는데, 그는 다음과 같은 내용이다.

1. 배열에서의 의미처럼 같은 TYPE 의 것이 여러개 뭉쳐진 형태를 나타내어야 한다.
2. 배열의 크기를 나타내는 tree node 는 여러 표현식의 복합적인 node 로 구성되어 있을 수 있다.
3. 배열의 크기가 지정이 안될 수 도 있다.
4. 배열의 크기는 음수가 아니다.

배열의 크기를 나타내는 표현식은

```
<integer_cst 0x4016c5c0 type <integer_type 0x40166380 int> constant 2>
```

이지만, 우리는 이것의 값을 2 가 아닌 1 로 변경해야 할 것이다. 이 크기를 나타내는 node 는 복잡한 표현식을 가지고 있을 수 있기 때문에, TREE AST 수준에서 처리가 이루어져야 하며, 이러한 것을 지원하기 위해 GCC 에서는 "folding"을 사용한다.

```
itype = fold (build (MINUS_EXPR, index_type,
             convert (index_type, size),
             convert (index_type, size_one_node)));
```

"folding" 을 지원하기 위해서 존재하는 fold () 함수의 경우 $prefix/gcc/fold-const.c 파일에 존재하게 된다. 이 후에, build_index_type () 함수를 통해서 index type 를 생성하게 되고, (여기서 index type 이란 배열의 크기를 가르키게 될 index 의 type 을 말한다.) 이 index type 과 기존 type 을 이용해서, 실제 배열 type 을 위한 node 를 build_array_type () 함수를 통해서 생성하게 된다. 그런 후 이제 build_decl () 를 통해서, VAR_DECL tree node 를 생성하게 되는 것이다. 이 과정내에서 아직 언급하지 않은 Type 혹은 Decl 에 대한 layout 을 수행하게 되는데, 이에 대한 부분 또한 다른 문서에서 언급해 나가겠다. 이렇게 해서 결과적으로 아래와 같은 node 가 만들어지게 된다.

```
<var_decl 0x401f6620 a
    type <array_type 0x401f65b0
        type <integer_type 0x401da380 int SI
            size <integer_cst 0x401d7540 constant 32>
            unit size <integer_cst 0x401d75e0 constant 4>
            align 32 symtab 0 alias set -1 precision 32
            min <integer_cst 0x401d75a0 -2147483648>
            max <integer_cst 0x401d75c0 2147483647>
            pointer_to_this <pointer_type 0x401e2620>>
        DI
        size <integer_cst 0x401d7900 constant 64>
        unit size <integer_cst 0x401d7b20 constant 8>
        align 32 symtab 0 alias set -1
        domain <integer_type 0x401df070
            type <integer_type 0x401de4d0 unsigned int> SI
            size <integer_cst 0x401d7540 32>
            unit size <integer_cst 0x401d75e0 4>
            align 32 symtab 0 alias set -1 precision 32
            min <integer_cst 0x401d7c00 0>
            max <integer_cst 0x401d7b60 1>>>
    public static DI file <stdin> line 1
    size <integer_cst 0x401d7900 64>
    unit size <integer_cst 0x401d7b20 8>
    align 32>
```

"folding"에 대해서는 다른 문서에서 이야기 하도록 하겠다.

- int abc ();

- declspecs

```
<tree_list 0x401e53c0 static
    value <identifier_node 0x401dd100 int tree_0
        global <type_decl 0x401daa80 int
            type <integer_type 0x401da380 int>
            VOID file <built-in> line 0
            align 1>
        rid 0x401dd100 "int">>
```

- declarator

```
<call_expr 0x401e05c0
    arg 0 <identifier_node 0x401f59c0 abc>
    arg 1 <tree_list 0x401e53fc>>
```

declarator 가 CALL_EXPR 이고 arg 1 에는 parameter 에 대한 node 가 오게 되는데, arg 1 은 grokparms () 함수를 통해서 지역 변수 arg_types 에 적당한 type 을 넣은 후, build_function_type () 함수를 통해서, 실제 함수에 적용할, type 을 생성하게 된다. 그런후 arg 0 을 declarator 로 재지정한다. 만들어진 type 은 아래와 같은 모양을 가지게 된다.

```
<function_type 0x401e28c0
    type <integer_type 0x401da380 int SI
        size <integer_cst 0x401d7540 constant 32>
        unit size <integer_cst 0x401d75e0 constant 4>
        align 32 symtab 0 alias set -1 precision 32
        min <integer_cst 0x401d75a0 -2147483648>
        max <integer_cst 0x401d75c0 2147483647>
        pointer_to_this <pointer_type 0x401e2620>>
    DI
    size <integer_cst 0x401d7900
        type <integer_type 0x401de540 bit_size_type> constant 64>
    unit size <integer_cst 0x401d7b20
        type <integer_type 0x401de4d0 unsigned int> constant 8>
    align 64 symtab 0 alias set -1>
```

이제, FUNCTION_TYPE 과, 새롭게 지정된 declarator 를 이용하여, build_decl () 를 통해서, FUNCTION_DECL node 를 생성하게 되며, 아래와 같은 모양을 가지게 된다. 물론 아래에 나와 있는 public, external 과 같은 설정은 전역 변수 current_binding_level 나 지역 변수 specbits 에 의해서 영향을 받아 설정되는 부분이다.

```
<function_decl 0x401f65b0 abc
    type <function_type 0x401e28c0
        type <integer_type 0x401da380 int SI
            size <integer_cst 0x401d7540 constant 32>
            unit size <integer_cst 0x401d75e0 constant 4>
            align 32 symtab 0 alias set -1 precision 32
            min <integer_cst 0x401d75a0 -2147483648>
            max <integer_cst 0x401d75c0 2147483647>
            pointer_to_this <pointer_type 0x401e2620>>
        DI
        size <integer_cst 0x401d7900 constant 64>
        unit size <integer_cst 0x401d7b20 constant 8>
        align 64 symtab 0 alias set -1>
    public external QI file <stdin> line 1>
```

- int abc (int a);

앞의 예제와 거의 비슷하지만, 내부에 parameter 가 실제로 선언되어 있는 경우인데, 처리 과정은 앞의 예제와 같지만, grokparms () 함수에서 반환되는 결과값과 build_function_type () 을 통해서 만들어진 내용은 위와 다르다고 할 수 있다.

- declspecs

```
<tree_list 0x401e53c0 static
    value <identifier_node 0x401dd100 int tree_0
        global <type_decl 0x401daa80 int
            type <integer_type 0x401da380 int>
            VOID file <built-in> line 0
            align 1>
        rid 0x401dd100 "int">>
```

- declarator

```
<call_expr 0x401e05c0
    arg 0 <identifier_node 0x401f59c0 abc>
    arg 1 <tree_list 0x401e549c
        purpose <parm_decl 0x401f6540 a
            type <integer_type 0x401da380 int>
            SI file <stdin> line 1
            size <integer_cst 0x401d7540 constant 32>
            unit size <integer_cst 0x401d75e0 constant 4>
            align 32 result <integer_type 0x401da380 int>
            initial <integer_type 0x401da380 int>
            arg-type <integer_type 0x401da380 int>
            arg-type-as-written <integer_type 0x401da380 int>>
        chain <tree_list 0x401e5474 value <integer_type 0x401da380 int>
            chain <tree_list 0x401e5488
                value <void_type 0x401de770 void>>>>>
```

Parameter 가 존재하기 때문에, arg 1 가 존재하게 되며, grokparms () 함수에 의해서 아래와 같이 주어진 PARM_DECL node 를 해석하게 된다.

```
<tree_list 0x401e5474
    value <integer_type 0x401da380 int SI
        size <integer_cst 0x401d7540 constant 32>
        unit size <integer_cst 0x401d75e0 constant 4>
        align 32 symtab 0 alias set -1 precision 32
        min <integer_cst 0x401d75a0 -2147483648>
        max <integer_cst 0x401d75c0 2147483647>
        pointer_to_this <pointer_type 0x401e2620>>
    chain <tree_list 0x401e5488
        value <void_type 0x401de770 void VOID
            align 8 symtab 0 alias set -1
            pointer_to_this <pointer_type 0x401de7e0>>>>
```

해석이 된 후, build_function_type () 함수를 통해서, 앞의 grokparms () 함수가 해석한 arg_types 과 조합하여, 결과적으로 FUNCTION_DECL node 를 완성하게 된다.

```
<function_decl 0x401f6620 abc
    type <function_type 0x401e32a0
        type <integer_type 0x401da380 int SI
            size <integer_cst 0x401d7540 constant 32>
            unit size <integer_cst 0x401d75e0 constant 4>
            align 32 symtab 0 alias set -1 precision 32
            min <integer_cst 0x401d75a0 -2147483648>
```

```
            max <integer_cst 0x401d75c0 2147483647>
            pointer_to_this <pointer_type 0x401e2620>>
        DI
        size <integer_cst 0x401d7900 constant 64>
        unit size <integer_cst 0x401d7b20 constant 8>
        align 64 symtab 0 alias set -1
        arg-types <tree_list 0x401d8adc value <integer_type 0x401da380 int>
            chain <tree_list 0x401d8974 value <void_type 0x401de770 void>>>>
    public external QI file <stdin> line 1>
```

5.1.2 초기화값의 존재 상태 확인

위의 하위 섹션에서, grokdeclarator () 함수의 operation 에 대해서 말하였다. 이제 이 함수의 수행이 끝난 후에, 발생되는 operation 에 대해 이야기 하도록 하겠다. start_decl () 함수를 통해서 들어온 initialized 인자에 따라 실행 여건이 달라지는데, 초기화값이 있을 경우, initialized 변수의 값은 1, 없을 경우 0 으로 세팅된다고 말하였다. 먼저 초기화값이 존재를 해도 괜찮은지 검사를 하게 된다. 검사하는 node 는 TYPE_DECL, FUNCTION_DECL, PARM_DECL 에 대해서 검사하고 불완전한 type 에 대해서도 초기화를 하지 않도록 수정한다.

그럼 결과적으로 finish_decl () 함수 혹은 pushdecl () 함수가 현재 선언하고 있는, 즉 grokdeclarator () 함수에 의해서 생성된 DECL 이 초기화될 것인지 아닌지를 어떻게 구분하는지에 대해서 알아보면,

```
DECL_INITIAL (decl) = error_mark_node;
```

위와 같이 error_mark_node 를 넣음으로써 'pushdecl' 함수에서 이것은 초기화된 decl 임을 말한다. 비록 아직 우리는 initializer expression 를 가지지 못했지만 말이다. 또한 'finish_decl' 함수에게 실제 초기화자를 저장할 것이라는 것을 말한다.

5.1.3 Attribute 의 설정

Attibute 의 설정은 decl_attributes () 함수에 의해서 설명되며, start_decl () 함수의 argument 로 들어온, attributes 를 이용하여 설정되게 된다. 이 함수의 설명은 아래와 같다.

> ATTRIBUTES 내에 list 되어 있는 attribute 들을 처리하고 그들을 *NODE 내에 설치합니다. *NODE 는 DECL (TYPE_DECL 을 포함) 혹은 TYPE 둘 중 하나를 가질 것입니다. 만약 DECL 이라면, 그자리에서 수정되어야 바람직할 것이다; 만약 TYPE 이라면, ATTR_FLAG_TYPE_IN_PLACE 가 FLAGS 에 설정되어있지 않은 이상, 복사본이 생성되어야 한다. FLAGS 는 추가적인 정보를 주며, tree.h 파일의 enum attribute_flags 를 bitwise OR 의 형태로 제공 한다. 이러한 flags 들을 기반으로, 몇몇 attribute 들이 나중에 있을 stage 에 적용되기 위해 반환되어질 수 있다. (예를 들면, decl attribute 를 그것의 type 보다 declaration 에 적용할 때) 만약, ATTR_FLAG_BUILT_IN 가 설정되어 있지 않고, *NODE 가 DECL 이면, 이 DECL 에 적용할 몇몇 default attribute 들이 있을 수 있는지 또한 고려한다; 만약 그렇다면, decl_attributes 는 이 attribute 들과 ATTR_FLAG_BUILT_IN set 들을 인자로 재귀적으로 호출될 것이다.

이 함수의 세번째 인자로 전달되는 enum attribute_flags 에 대해서 잠시 살펴보면 아래와 같은 내용이다.

```
enum attribute_flags
{
  ATTR_FLAG_DECL_NEXT = 1,
  ATTR_FLAG_FUNCTION_NEXT = 2,
  ATTR_FLAG_ARRAY_NEXT = 4,
  ATTR_FLAG_TYPE_IN_PLACE = 8,
  ATTR_FLAG_BUILT_IN = 16
};
```

각 요소에 대해서 설명을 하면, 아래와 같은 내용이다.

- ATTR_FLAG_DECL_NEXT

  전달되는 type 이 DECL 의 type 이며, type 이 반환되기 보다 DECL 에 적용하기 위해 다시 전달되어져야 하는 어떤 attribute 들이다.

- ATTR_FLAG_FUNCTION_NEXT

  전달되는 type 이 function return type 이며, return type 이 반환되기 보다 function type 에 적용하기위해 다시 전달되어져야 하는 어떤 attribute 들이다.

- ATTR_FLAG_ARRAY_NEXT

  전달되는 type 이 array element type 이며, element type 이 반환되기 보다 array type 에 적용하기위해 다시 전달되어져야 하는 어떤 attribute 들이다.

- ATTR_FLAG_TYPE_IN_PLACE

  전달되는 type 은 생성될 structure 혹은 union, enumeration type 이며, 그 자리에서 수정되어야 한다.

- ATTR_FLAG_BUILT_IN

  attribute 들은 기본적으로 library function 의 이름이 known behavior 를 가르키는 것에 적용되는 중이며, 만약 그것이 사실 function type 과 호환이 되지 않을 경우 조용히 무시되어 져야 한다.

5.1.4 Binding Level 에 ..._DECL node 의 추가

새로운 ..._DECL node 를 지금까지 생성하고, attribute 까지 적용하였다. 그렇게 생성된 node 를 현재의 binding level 에 등록해야 하는 일이 마지막으로 남아 있다. 이러한 operation 은 pushdecl () 함수에 의해서 이루어지는데, 이에 대해서 이제 알아보도록 하자.

pushdecl () 함수를 통해 수행을 하면, 최종적으로 현재 혹은 Global binding level 의 names 에 현재 선언중인 ..._DECL 을 등록하게 된다. 만약 전역 변수로써 “int a;” 를 선언할 경우, 다음과 같은 구성을 보일 것이다. 즉 binding level 의 b 라고 했을 때, b→names 의 구성은 아래와 같게 된다.

```
<var_decl 0x401f6540 a
    type <integer_type 0x401da380 int SI
        size <integer_cst 0x401d7540 constant 32>
        unit size <integer_cst 0x401d75e0 constant 4>
        align 32 symtab 0 alias set -1 precision 32
        min <integer_cst 0x401d75a0 -2147483648>
        max <integer_cst 0x401d75c0 2147483647>
        pointer_to_this <pointer_type 0x401e2620>>
    public static common SI file <stdin> line 17
    size <integer_cst 0x401d7540 32> unit size <integer_cst 0x401d75e0 4>
    align 32 chain <type_decl 0x401eed90 __g77_ulongint>>
```

names 의 목록 list 는 TREE_CHAIN (x) 에 의해서 이루어지게 되며, tree 의 common.chain 을 통해서 서로 연결되게 된다. 이 ..._DECL 를 넣을 때 반대 순서로 list 에 decl 들을 놓는데, 새로운 것이 앞에 오게 된다. 이것은 나중에 필요하다면 우리는 그것을 다시 거꾸로 할 것이다.

하지만 pushdecl () 함수에서는 현재 binding level 의 names 에 ..._DECL node 만 넣는 것은 아닌데, 전역 변수 혹은 지역 변수, 함수 등등 들어오는 ..._DECL 에 대한 DECL_NAME (x) 를 처리하게 된다. 대부분의 이름은 IDENTIFIER_NODE tree node 를 통해서 구성되어 있으며, 앞의 문서에서 이미 언급했듯이, 실제 identifier_node tree node 에 필요한 구조체 크기 보다 약간 더 큰 공간이 할당되어 있는데, 즉 struct lang_identifier 구조체 구성요소를 이루는 element 만큼 더 할당된다고, 앞에서 말하였다.

lookup_name_current_level () 함수를 통해서, 해당 값을 구하여서 IDENTIFIER_NODE 를 위해 적당한 설정을 하게 된다.

예를 들어, 전역 변수 "int a;"에 대해 말을 하면, 일반적으로 이러한 처리가 있기 전 IDENTIFIER_NODE tree node 의 경우, 아래와 같은 모습을 하게 되는데,

```
<identifier_node 0x401f59c0 a>
```

처리가 있은 후 아래와 같이 변하게 된다.

```
<identifier_node 0x401f59c0 a public
    global <var_decl 0x401f6540 a
        type <integer_type 0x401da380 int SI
            size <integer_cst 0x401d7540 constant 32>
            unit size <integer_cst 0x401d75e0 constant 4>
            align 32 symtab 0 alias set -1 precision 32
            min <integer_cst 0x401d75a0 -2147483648>
            max <integer_cst 0x401d75c0 2147483647>
            pointer_to_this <pointer_type 0x401e2620>>
        public static common SI file <stdin> line 17
        size <integer_cst 0x401d7540 32> unit size <integer_cst 0x401d75e0 4>
        align 32>>
```

이러한 처리가 있는 이유는 각 identifier 가 어떠한 선언으로 만들어진 것인지 알려주기 위해서이다. 만약 "int main () " 와 같이 간단한 함수 선언을 할 경우 아래와 같이 main IDENTIFIER_NODE 가 설정되게 된다.

```
<identifier_node 0x401ed6c0 main public
    global <function_decl 0x401f65b0 main
        type <function_type 0x401e28c0 type <integer_type 0x401da380 int>
            DI
            size <integer_cst 0x401d7900 constant 64>
            unit size <integer_cst 0x401d7b20 constant 8>
            align 64 symtab 0 alias set -1>
        public static QI file <stdin> line 15 initial <error_mark 0x401dca20>>>
```

5.2 초기화가 없는 변수 정의 (finish_decl)

앞에서 이야기 했듯이, finish_decl () 함수에서는 앞에서 선언한 ....DECL node 가 불완전한 type 을 가지고 있을 경우, 이 함수에서 초기화값을 보고 그에 대해 정의를 하게 되어 있다. 불완전한 type 인지 아닌지를 가리는 부분은 DECL_SIZE (decl) 가 설정되어 있느냐 아니냐에 달려 있는데, 이러한 부분을 설정하는 부분이 layout 부분이고, 불완전한 ....DECL node 의 경우, layout 이 실행되지 않았음을 의미한다. finish_decl () 함수는 또한 초기값(초기화자) 를 설치한다. 또 만약 배열 type 의 길이가 앞에서 알려지지 않았다면 그것에 대해 초기값으로부터 알아내어 이제 선언해야만 한다. 만약 그렇지 못할 경우 오류가 된다.

전역 변수인지, 지역 변수인지에 대한 구분은 DECL_CONTEXT (decl) 가 설정되어 있는냐의 문제인데, 그 이유는 pushdecl () 함수에서 아래와 같은 operation 이 일어나기 때문이다.

```
DECL_CONTEXT (x) = current_function_decl;
```

전역 변수일 경우, rest_of_decl_compilation () 함수가 수행되어 처리되며, 지역 변수일 경우, add_decl_stmt () 함수가 호출될 수 있다.

5.3 초기값이 있는 변수 정의

이제 해당 변수의 초기화자 (initializer) 가 존재할 경우, GCC 에서는 어떻게 처리되는지 알아보도록 하자. 앞의 문서에서 초기화자가 해석되기 전에, start_init () 함수가 호출되고, finish_decl () 함수를 통해, 변

수의 선언을 끝내기 전에, finish_init () 함수를 호출한다는 것을 언급하였다. 초기화자를 저장하기 위해서, GCC 에서는 struct initializer_stack 구조체를 이용하여 stack 형태로 저장하게 되는데, 우선 이 구조체에 대해 먼저 알아보도록 하자.

```
struct initializer_stack
{
  struct initializer_stack *next;
  tree decl;
  const char *asmspec;
  struct constructor_stack *constructor_stack;
  struct constructor_range_stack *constructor_range_stack;
  tree elements;
  struct spelling *spelling;
  struct spelling *spelling_base;
  int spelling_size;
  char top_level;
  char require_constant_value;
  char require_constant_elements;
  char deferred;
};
```

이 구조체 stack 은 nest 된 분리된 initializer 들을 기록한다. Nest 된 initializer 들은 ANSI C 에서는 발생할 수 없지만, GNU C 에서는 { ... (struct foo) { ... } ... } 와 같은 경우 허락한다. 다음으로 이 구조체가 포함하고 있는 다른 구초제에 대해서 알아보도록 하자.

```
struct constructor_stack
{
  struct constructor_stack *next;
  tree type;
  tree fields;
  tree index;
  tree max_index;
  tree unfilled_index;
  tree unfilled_fields;
  tree bit_index;
  tree elements;
  struct init_node *pending_elts;
  int offset;
  int depth;
  /* If nonzero, this value should replace the entire
     constructor at this level.  */
  tree replacement_value;
  struct constructor_range_stack *range_stack;
  char constant;
  char simple;
  char implicit;
  char erroneous;
  char outer;
  char incremental;
  char designated;
};
```

이 stack 은 가장 외곽의 것을 포함하여, 각각 초기화자를 만드는 implicit 혹은 explicit level 용 level 을 가지고 있다. 이것은 위 변수들의 대부분의 값들을 저장한다.

```
struct constructor_range_stack
{
  struct constructor_range_stack *next, *prev;
  struct constructor_stack *stack;
  tree range_start;
  tree index;
  tree range_end;
  tree fields;
};
```

이 stack 은 list 내 몇몇 range designator 부터 마지막 designator 까지 designator 들을 나타낸다.

```
struct spelling
{
  int kind;
  union
    {
      int i;
      const char *s;
    } u;
};
```

이름의 component 들이 push 혹은 pop 될 수 있도록 하는 spelling stack 를 수행한다. stack 상의 각 element 는 이 구조체이다. 각 구성요소에 대해 설명은 하면 아래와 같다.

- kind

  Kind 는 세 가지 종류의 값을 가지며, 아래와 같은 값이 존재할 수 있다.

  ```
  #define SPELLING_STRING 1
  #define SPELLING_MEMBER 2
  #define SPELLING_BOUNDS 3
  ```

- union u

  실제 값이 들어갈 곳이다.

그 외, 이러한 spelling stack 을 위한 MACRO 가 다수 존재하는데, PUSH_SPELLING, SAVE_SPELLING_DEPTH 와 같은 macro 들을 $prefix/gcc/c-typeck.c 파일에서 살펴보길 바란다.

그럼 실제로 위의 구조체에 어떠한 값들이 저장되게 되는지 알아보도록 하자. 실제로 저장하는 부분은 start_init () 함수에 저장되어 있는데, struct initializer_stack 구조체에 대한 pointer 인 p 가 있다고 가정했을 때, 실제 값은 아래와 같이 설정되게 된다.

```
  p->decl = constructor_decl;
  p->asmspec = constructor_asmspec;
  p->require_constant_value = require_constant_value;
  p->require_constant_elements = require_constant_elements;
  p->constructor_stack = constructor_stack;
  p->constructor_range_stack = constructor_range_stack;
  p->elements = constructor_elements;
  p->spelling = spelling;
  p->spelling_base = spelling_base;
  p->spelling_size = spelling_size;
  p->deferred = constructor_subconstants_deferred;
  p->top_level = constructor_top_level;
  p->next = initializer_stack;
```

위에서 설정되는 값들은 모두 전역변수로써 $prefix/gcc/c-typeck.c 파일에 선언되어 있다. 각각에 대한 설명을 하면 아래와 같다.

- constructor_decl

  initializer 가 읽어지고 있는 DECL node. 값이 0 일 경우, 우리가 (struct foo) ... 와 같은 constructor expression 를 읽고 있음을 의미한다.

- constructor_asmspec

  start_init 는 really_start_incremental_init 를 위해 여기 ASMSPEC arg 를 저장한다.

- require_constant_value

  아직 정확한 설명이 없음.

- require_constant_elements

  아직 정확한 설명이 없음.

- constructor_stack

  아직 정확한 설명이 없음.

- constructor_range_stack

  아직 정확한 설명이 없음.

- constructor_elements

  만약 우리가 element 들을 할당하기 보다, 절약하고자 한다면. 이것은 지금까지 element 들의 list (Reverse 순서이며, 가장 최근이 먼저 온다.) 이다.

- spelling

  (사용 안된) 다음 stack element.

- spelling_base

  Spelling stack base.

- spelling_size

  Spelling stack 의 크기.

- constructor_subconstants_deferred

  만약 defer_addressed_constants 가 호출된 적이 있다면 1.

- constructor_top_level

  만약 이것이 top-level decl 를 위한 initializer 일 경우 0 이 아니다.

- initializer_stack

  struct initializer_stack linked-list 의 처음을 가르키는 pointer.

그외에 추가적으로 start_init () 함수내에서 초기화가 이루어지는 전역변수가 또한 존재하는데, 그에 대해서 간단히 설명을 하면 아래와 같다.

- constructor_designated

이 initializer 에 어떠한 member designator 들이 존재할 경우 0 이 아닌 값.

- missing_braces_mentioned

  만약 우리가 이미 이 초기화자 내에 "missing braces around initializer" 메세지를 출력한 적이 있을 경우, 0 이 아닌 값을 가짐.

전역변수 constructor_decl, constructor_asmspec, constructor_top_level 는 각각 start_init () 함수가 호출될 때 전달된 argument 로 설정이 되며, require_constant_value 와 require_constant_elements 는 각 해당 ..._DECL node 의 static 및 TREE_CODE 에 따라 값이 달라진다. 그 외의 나머지 전역 변수는 모두 0 혹은 NULL 로 설정되게 된다. 그리고 현재 ..._DECL node 의 IDENTIFIER_POINTER 가 존재할 경우, 이를 위한 push_string () 함수가 호출되며, 호출된 이 함수에서는 struct spelling 에 해당 이 함수의 이름을 기록하게 된다.

이제 finish_init () 함수가 호출되면, 해당 initializer 의 모든 constructor stack 을 free 하고 그 외 나머지 operation 은 아래에 있는 것이 전부이다.

```
constructor_decl = p->decl;
constructor_asmspec = p->asmspec;
require_constant_value = p->require_constant_value;
require_constant_elements = p->require_constant_elements;
constructor_stack = p->constructor_stack;
constructor_range_stack = p->constructor_range_stack;
constructor_elements = p->elements;
spelling = p->spelling;
spelling_base = p->spelling_base;
spelling_size = p->spelling_size;
constructor_subconstants_deferred = p->deferred;
constructor_top_level = p->top_level;
initializer_stack = p->next;
free (p);
```

실제로 초기화값이 ..._DECL node 에 적용되는 것은 finish_decl () 함수가 호출되어질 때라고 말하였다. 그중 TREE_CODE (decl) 가 TYPE_DECL 가 아닌 모든 경우, store_init_value () 함수에서 실제 ..._DECL node 에 초기화값을 반영하게 된다. 이 함수는 변수의 초기값에 적당한 변환을 수행한 후 declaration DECL 내 저장하고 적당한 어떤 오류 메세지들을 출력한다. 만약 init 가 유효하지 않다면, ERROR_MART 을 저장한다. 결과적으로는 DECL_INITIAL (decl) 에 값을 넣게 된다는 것을 앞에서 말하였다.

제 6 절 함수

실제로 함수를 처리할 때 어떠한 일이 발생하게 되는지에 대해 아래부터 살펴보도록 하겠다. 함수의 경우, 크게 네 부분으로 구분할 수 있겠는데, 첫번째는 type specifier, 두번째는 함수의 이름, 세번째는 argument, 네번째는 함수의 body 일 것이다. type specifier 의 경우 앞에서 언급한 "Type 의 정의"에서 벗어나는 내용이 아니기 때문에, 이에 대한 설명은 필요하지 않으리라 본다.

6.1 인자

함수의 경우, 처음 정의를 할 때나, 실제로 다른 함수 내에서 호출을 할 경우, 옆에 parameter 들을 받아들이게 되는데, 이 부분에 대한 처리는 어떻게 되는지 이 하위섹션에서 살펴보자.

실제 함수의 인자를 처리하는 Yacc 분법은 parmlist_or_identifiers 혹은 parmlist 라벨에서 처리한다고 할 수 있다. Parameter 들을 처리하기 전에 몇가지 준비 작업을 하는 것이 있다.

- 새로운 binding level 을 생성한다. pushlevel () 함수에서 수행된다. Binding level 에 대해서는 아래 부분에서 살펴보도록 하자.

- 새로이 만들어진 binding level 의 구성 요소인 parm_order 를 NULL_TREE 로 초기화한다. clear_parm_order () 함수에서 수행된다.
- 이 binding level 이 parameter 를 위한 level 로 사용된다는 사실을 가르키도록 구성 요소인 parm_flag 를 1 로 설정한다. declare_parm_level () 함수에서 수행된다.

이렇게 준비가 된 후, (...) 사이에 선언된 각 변수를 해석하게 된다. 만약 Old-style 의 parameter 선언일 경우, 다른 변수 선언과 마찬가지로 처리되므로 위에서 설명한 "변수" 섹션을 참고할 수 있을 것이다. 물론 (...) 사이에 Type 과 Declrator 도 물론 거의 다르지 않다. 차이점이 있다면, 각 Type 과 Declarator 를 build_tree_list () 함수를 통해 하나의 node 로 묶은 후에 push_parm_decl () 함수를 호출하게 된다는 것이다. 만약 Parameter 에 int a 라고 선언했다면, 만들어지는 node 는 아래와 같다.

```
<tree_list 0x401e5438
    purpose <tree_list 0x401e53fc static
        value <identifier_node 0x401dd100 int tree_0 global <type_decl 0x401daa80 int>
            rid 0x401dd100 "int">>
    value <identifier_node 0x401f5a00 a>>
```

만약 char *argv[] 와 같이 선언하였다면, 아래와 같은 모형을 가지게 된다.

```
<tree_list 0x401e544c
    purpose <tree_list 0x401e53fc static
        value <identifier_node 0x401d9d80 char tree_0 global <type_decl 0x401daaf0 char>
            rid 0x401d9d80 "char">>
    value <indirect_ref 0x401e5438
        arg 0 <array_ref 0x401e05c0
            arg 0 <identifier_node 0x401f5a00 argv>>>>
```

이렇게 만들어진 Tree node 는 push_parm_decl () 함수에 의해서 처리가 되게 되는데, 이 함수는 주어진 parsed parameter declaration 를 PARM_DECL 로 해석하고 현재 binding level 에 push 하는 역활을 하며, 또한 앞으로 parm decls 의 이득을 위해 현재 binding level 의 'parm_order' 에 주어진 parm 들의 순서를 기록하게 된다. 물론 변수를 선언할 때와 마찬가지로, PARM_DECL 를 만들기 위해, grokdeclarator () 함수를 호출하고, 실제로 변수를 현재의 Binding level 에 반영하기 위해, pushdecl () 함수가 호출되며, 마지막으로 finish_decl () 함수가 호출되게 된다.

이제 push_parm_decl () 의 operation 이 끝났다면, get_parm_info () 함수가 호출되게 된다. 이 부분에서의 처리 과정은 다음과 같다. 현재 binding level 에는 선언했던 변수에 대한 'names' 정보가 기록되어 있을 경우, 그리고 'parm_order' 에는 변수의 순서가 기록되어 있을 것인데, 이것은 역순으로 기록되어 있을 것이다. 만약 (int a, char *ar[]) 이라는 parameter 를 넣었다면 다음과 같은 모습으로 'parm_order' 에 기록되어 있을 것이다.

```
<tree_list 0x401e54ec
    value <parm_decl 0x401f6690 ar
        type <pointer_type 0x401f65b0 type <pointer_type 0x401dec40>
            unsigned SI
            size <integer_cst 0x401d7b80 constant 32>
            unit size <integer_cst 0x401d7be0 constant 4>
            align 32 symtab 0 alias set -1>
        unsigned SI file <stdin> line 1
        size <integer_cst 0x401d7b80 32> unit size <integer_cst 0x401d7be0 4>
        align 32 result <pointer_type 0x401f65b0>
        initial <pointer_type 0x401f65b0> arg-type <pointer_type 0x401f65b0>
        arg-type-as-written <pointer_type 0x401f65b0>
        chain <parm_decl 0x401f6540 a type <integer_type 0x401da380 int>
            SI file <stdin> line 1
            size <integer_cst 0x401d7540 constant 32>
```

```
            unit size <integer_cst 0x401d75e0 constant 4>
            align 32 result <integer_type 0x401da380 int>
            initial <integer_type 0x401da380 int>
            arg-type <integer_type 0x401da380 int>
            arg-type-as-written <integer_type 0x401da380 int>>>
    chain <tree_list 0x401e5460 value <parm_decl 0x401f6540 a>>>
```

이 정보를 바탕으로 해서, 실제 'names' 정보를 아래와 같이 순서에 맞게 재정리해 놓는다. 이를 위해서 nreverse () 함수를 사용하는데, 단순히 참조하기 바란다.

```
<parm_decl 0x401f6540 a
    type <integer_type 0x401da380 int SI
        size <integer_cst 0x401d7540 constant 32>
        unit size <integer_cst 0x401d75e0 constant 4>
        align 32 symtab 0 alias set -1 precision 32
        min <integer_cst 0x401d75a0 -2147483648>
        max <integer_cst 0x401d75c0 2147483647>
        pointer_to_this <pointer_type 0x401e2620>>
    SI file <stdin> line 1 size <integer_cst 0x401d7540 32>
    unit size <integer_cst 0x401d75e0 4>
    align 32 result <integer_type 0x401da380 int>
    initial <integer_type 0x401da380 int>
    arg-type <integer_type 0x401da380 int>
    arg-type-as-written <integer_type 0x401da380 int>
    chain <parm_decl 0x401f6690 ar>>
```

새롭게 정리가 끝나면, 'names' 정보를 새롭게 업데이트를 하고, 반환활 TREE_LIST node 를 만들게 되는데, 위의 예를 계속 든다면, 아래와 같은 node 가 만들어져 반환되게 된다. Purpose 에는 해당 PARM_DECL node 의 chain 이 Chain 에는 해당 type 의 chain 이 들어가게 된다.

```
<tree_list 0x401e553c
    purpose <parm_decl 0x401f6540 a type <integer_type 0x401da380 int>
        SI file <stdin> line 1
        size <integer_cst 0x401d7540 constant 32>
        unit size <integer_cst 0x401d75e0 constant 4>
        align 32 result <integer_type 0x401da380 int>
        initial <integer_type 0x401da380 int>
        arg-type <integer_type 0x401da380 int>
        arg-type-as-written <integer_type 0x401da380 int>
        chain <parm_decl 0x401f6690 ar>>
    chain <tree_list 0x401e5500 value <integer_type 0x401da380 int>
        chain <tree_list 0x401e5514 value <pointer_type 0x401f65b0>
            chain <tree_list 0x401e5528 value <void_type 0x401de770 void>>>>>
```

이렇게 처리된 node 는 상위 Yacc 문법의 라벨에게 넘기게 되며, parmlist_tags_warning () 함수를 호출하여, 완료될 수 없는 struct 혹은 union, enum tag 들에 관해 경고를 한다. 그리고 poplevel () 함수를 통해서 현재 binding level 에서 탈출하게 된다. poplevel () 함수에 대한 자세한 설명은 아래의 섹션 "Binding Level" 을 참조하기 바란다.

이렇게 처리가 완료가 되면, 위의 node 는 CALL_EXPR 과 결합하여 start_function () 함수에 전달되며, 이 함수를 통해서 몇몇 전역 변수와 FUNCTION_DECL node 가 생성 및 설정된다. 그런후 start_function () 함수가 완료가 된 후, store_parm_decls () 함수가 호출되게 된다. start_function () 함수에 대해서는 아래에서 설명한 "함수의 시작 (start_function)" section 을 참조하기 바란다.

그럼 아래부터는 현재 함수 선언내에 parameter 선언들을 저장하는데 사용되는 store_parm_decls () 함수에 대해서 알아보도록 하자. 이 함수의 경우, 이 외에도 좀 더 많은 일을 내부에서 수행을 하고 있는데,

함수를 위한 RTL code 를 초기화 한다거나, statement tree 를 시작하는 일 등등 을 하게 된다. 즉, 실제로 function 의 body 를 해석하기 전에 이루어져야 할 일을 마무리 짓는 성격을 같이 가진다.

우선 이 함수의 본래의 목적인 current_function_decl 에 parameter 선언을 저장하는 부분을 살펴보도록 하자. C 언어에서는 두 가지 prototype 을 정의해 주는 것이 가능하기 때문에, ANSI prototype 혹은 old-style definition 중 어떤 것으로 정의해 주었느냐에 따라서 수행 방법이 달라지게 된다. 아래에서는 ANSI prototype 형식으로 정의를 하였을 때를 살펴보도록 하자.

ANSI prototype 의 경우, Parm 들이 이미 decl 들을 가지고 있어서, 실제로 그들을 적용하기 위해 기록하는 것을 제외하곤 할 일은 없다. 각 PARM_DECL 에 대해 pushdecl () 를 실행 한 후, 원래 chain order 형태로 decl 를 얻어 funciton 에 기록한다. 즉 아래와 같은 수행을 통해서 parameter 를 저장하게 된다.

```
DECL_ARGUMENTS (fndecl) = getdecls ();
```

그리고 enum constant 들은 pushdecl 한후, tags 들을 저장함으로써, 저장 부분을 완료하게 된다.

```
int
main (int argc, char *argv[]) {
}
```

를 예제로, 여기까지 완료되었을 때, 실제, FUNCTION_DECL 의 TREE node 를 살펴본다면, 아래와 같은 모습을 하게 된다.

```
<function_decl 0x401f6770 main
   type <function_type 0x401f6700
       type <integer_type 0x401da380 int SI
           size <integer_cst 0x401d7540 constant 32>
           unit size <integer_cst 0x401d75e0 constant 4>
           align 32 symtab 0 alias set -1 precision 32
           min <integer_cst 0x401d75a0 -2147483648>
           max <integer_cst 0x401d75c0 2147483647>
           pointer_to_this <pointer_type 0x401e2620>>
       DI
       size <integer_cst 0x401d7900 constant 64>
       unit size <integer_cst 0x401d7b20 constant 8>
       align 64 symtab 0 alias set -1
       arg-types <tree_list 0x401e5500 value <integer_type 0x401da380 int>
           chain <tree_list 0x401e5514 value <pointer_type 0x401f65b0>
               chain <tree_list 0x401e5528 value <void_type 0x401de770 void>>>>>
   public static QI file <stdin> line 2
   arguments <parm_decl 0x401f6540 argc type <integer_type 0x401da380 int>
       SI file <stdin> line 2 size <integer_cst 0x401d7540 32>
       unit size <integer_cst 0x401d75e0 4>
       align 32 context <function_decl 0x401f6770 main>
       result <integer_type 0x401da380 int>
       initial <integer_type 0x401da380 int>
       arg-type <integer_type 0x401da380 int>
       arg-type-as-written <integer_type 0x401da380 int>
       chain <parm_decl 0x401f6690 argv type <pointer_type 0x401f65b0>
           unsigned SI file <stdin> line 2
           size <integer_cst 0x401d7b80 constant 32>
           unit size <integer_cst 0x401d7be0 constant 4>
           align 32 context <function_decl 0x401f6770 main>
           result <pointer_type 0x401f65b0>
           initial <pointer_type 0x401f65b0>
           arg-type <pointer_type 0x401f65b0>
```

```
        arg-type-as-written <pointer_type 0x401f65b0>>>
    result <result_decl 0x401f67e0 type <integer_type 0x401da380 int>
        SI file <stdin> line 2 size <integer_cst 0x401d7540 32>
        unit size <integer_cst 0x401d75e0 4>
        align 32> initial <error_mark 0x401dca20>
    (mem:QI (symbol_ref:SI ("main")) [0 S1 A8])
    chain <type_decl 0x401eed90 __g77_ulongint>>
```

이 부분까지 완료가 되었다면, 실제로 FUNCTION_DECL 에 대한 설정은 중반 부분까지 완료가 되었다고 할 수 있는데, store_parm_decls () 함수의 나머지 부분에서는 init_function_start () 함수 및 begin_stmt_tree () 함수를 실행하게 되며, 실제 함수 구조체인 struct function 을 위한 몇몇 처리가 이루어지게 된다. 이 부분에 대해서는 이 부분에서 언급하지 않겠고, 아래의 "함수의 중간" subsection 을 보기 바란다.

6.2 함수의 시작 (start_function)

함수의 정의를 처리하는 과정에서 old_style_parm_decls (Yacc 문법) 라벨에 도달하기 전에, GCC 에서는 함수의 시작을 알리기 위해 start_function () 함수를 호출하게 되며, 인자로써, 현재 함수가 선언되고 있는 Type, 즉 current_declspecs, 바로 앞 단에서 설명한 Declarator (물론 parm list 를 포함한), 그리고 현재 함수에 적용될 Attribute 들, all_prefix_attributes 를 전달하게 된다.

현재 type 과, attribute 의 경우, 앞서 설명한 변수의 선언과 그의 같은, setspecs 라벨에 의해서 기록되는 값이기 때문에 이에 대해서는 설명하지 않겠지만, declarator 에 대해서는 앞 단에서 구한 parm list node 와 함수의 이름을 위한 identifier_node 가 CALL_EXPR 로 만들어져야 하기 때문에 이에 대해 설명을 약간 하겠다.

앞의 "인자" 부분에서 get_parm_info () 함수가 주었던 node 와 실제 함수의 이름을 가르키는 identifier_node 를 build_nt () 함수를 이용하여 CALL_EXPR tree node 를 만들게 되는데,
이 node 가 start_function 에 건네지게 되며, 이를 이용하여 FUNCTION_DECL tree node 가 새롭게 만들어지게 된다. 결과적으로 만들어지는 CALL_EXPR node 의 경우, 아래와 같은 모형을 가지게 된다.

```
<call_expr 0x401e05e0
    arg 0 <identifier_node 0x401ed6c0 main>
    arg 1 <tree_list 0x401e553c
        purpose <parm_decl 0x401f6540 argc type <integer_type 0x401da380 int>
            SI file <stdin> line 1
            size <integer_cst 0x401d7540 constant 32>
            unit size <integer_cst 0x401d75e0 constant 4>
            align 32 result <integer_type 0x401da380 int>
            initial <integer_type 0x401da380 int>
            arg-type <integer_type 0x401da380 int>
            arg-type-as-written <integer_type 0x401da380 int>
            chain <parm_decl 0x401f6690 argv>>
        chain <tree_list 0x401e5500 value <integer_type 0x401da380 int>
            chain <tree_list 0x401e5514 value <pointer_type 0x401f65b0>
                chain <tree_list 0x401e5528 value <void_type 0x401de770 void>>>>>>
```

그럼 이제부터 본격적으로 start_function () 함수에서 발생하는 처리 부분에 대해서 보게 되겠는데, 하나 하나 설명하기에는 모호한 부분이 있기 때문에, 예제를 놓고 그 예제가 어떻게 처리되는지에 대해 살펴보도록 하자. 예제로는 아래와 같이 C 언어의 가장 대표적인 함수에 대해 알아보겠다.

```
int
main (int argc, char *argv[]) {
}
```

우선 이렇게 함수가 정의되었을 때, start_function () 에게 전달되는 인자들부터 살펴보도록 하자. declarator 의 경우, 바로 위에서 보인 CALL_EXPR node 모양을 갖게 된다. declspecs 의 경우, 아래와 같은 모형을 가지게 된다.

```
<tree_list 0x401e53c0 static
    value <identifier_node 0x401dd100 int tree_0
        global <type_decl 0x401daa80 int type <integer_type 0x401da380 int>
            VOID file <built-in> line 0
            align 1>
        rid 0x401dd100 "int">>
```

그리고 attributes 는 정의된 것이 없기 때문에, NULL 값을 가지고 있다. 우선 start_function 에서는 전역 변수 몇 개를 0 으로 초기화 시킨다.

```
  current_function_returns_value = 0;
  current_function_returns_null = 0;
  current_function_returns_abnormally = 0;
  warn_about_return_type = 0;
  current_extern_inline = 0;
  c_function_varargs = 0;
  named_labels = 0;
  shadowed_labels = 0;
  immediate_size_expand = 0;
```

위와 같은 것은 내용인데, 각 변수의 설명은 아래와 같다.

- current_function_returns_value

  함수 정의의 처음에는 이것을 0 으로 설정합니다. 만약 return 값을 나타내는 return statement 가 보일 경우 1 로 설정합니다.

- current_function_returns_null

  함수 정의의 처음에는 이것을 0 으로 설정합니다. 만약 인자가 없는 return statement 가 보일 경우 1 로 설정합니다.

- current_function_returns_abnormally

  함수 정의의 처음에는 이것을 0 으로 설정합니다. 만약 noreturn 함수로의 호출 (call) 이 보일 경우 1 로 설정합니다.

- warn_about_return_type

  Return type 이 기본(defaluted)으로 설정된 함수를 'grokdeclarator'가 처리할 때 이에 대한 경고 메세지의 발생이 요구된다면 값이 0 이 아닌 것으로 설정합니다.

- current_extern_inline

  함수 선언이 'extern inline' 으로 시작할 경우 0 이 아닌 값을 가집니다.

- c_function_varargs

  store_parm_decls 가 호출되어질 때, 값이 0 이 아님은 이것이 varargs function 임을 가르킨다. store_parm_decls 가 호출된 후의 값의 변화는 의미가 없다.

- named_labels

이름을 가지고 있는 함수내에서의 모든 LABEL_DECL 들의 list (TREE_LIST node 들의 chain). 이렇게 함으로써 우리는 함수의 끝에서 그들 이름들의 정의들을 깨끗히 할 수 있습니다.

- shadowed_labels

  현재 shadow 되어져 있는 outer context 들로부터 LABEL_DECL 들의 list.

- immediate_size_expand

  0 이 아닐 경우 이제 안전하게 expand_expr 를 호출할 수 없음을 의미합니다. 그래서 대신에 'pending_sizes' 상에 변수 크기들을 놓습니다.

초기화가 완료된 후에, declarator 와 declspecs 를 인자로 하여, grokdeclarator () 함수를 호출하게 된다. 물론 이 함수는 tree node 들을 해석해서 적당한 FUNCTION_DECL 을 반환하게 된다. 물론 입력된 인자가 이상할 경우 function definition 에 적당하지 않아, syntax error 를 발생시킬 것이다. 또한 declarator 가 해석되는 동안 전역변수 last_function_parms, last_function_parm_tags 는 이 함수 내부에서 호출되는 grokparms () 함수에 의해서 새롭게 설정될 것이다. grokdeclarator () 함수를 거쳐 생성된 tree node 는 아래와 같이 변하게 된다.

```
<function_decl 0x401f6770 main
    type <function_type 0x401f6700
        type <integer_type 0x401da380 int SI
            size <integer_cst 0x401d7540 constant 32>
            unit size <integer_cst 0x401d75e0 constant 4>
            align 32 symtab 0 alias set -1 precision 32
            min <integer_cst 0x401d75a0 -2147483648>
            max <integer_cst 0x401d75c0 2147483647>
            pointer_to_this <pointer_type 0x401e2620>>
        DI
        size <integer_cst 0x401d7900 constant 64>
        unit size <integer_cst 0x401d7b20 constant 8>
        align 64 symtab 0 alias set -1
        arg-types <tree_list 0x401e5500 value <integer_type 0x401da380 int>
            chain <tree_list 0x401e5514 value <pointer_type 0x401f65b0>
                chain <tree_list 0x401e5528 value <void_type 0x401de770 void>>>>>
    public external QI file <stdin> line 2>
```

이제는 declarator 와 declspecs 대신에, 위 node 를 사용하게 될 것이다. 그리고 grokdeclarator () 함수의 실행이 완료된 후 설정되는 전역 변수를 넣게 되는데,

```
current_function_parms = last_function_parms;
current_function_parm_tags = last_function_parm_tags;
```

는 store_parm_decls () 함수가 이 함수의 declarator 로부터 parm names 와 decl 를 찾을 수 있도록 저장한다. store_parm_decls () 함수는 start_function () 함수가 완료되고, Old-style 의 parameter 해석을 마찬후에 실행되는 함수이다. 그리고 위의 last_... 변수는 grokparms () 함수에서 설정된다고 앞에서 말하였다. 잠시, 각 전역변수에 대해 설명을 하면 아래와 같다.

- current_function_parms

  function definition 를 시작하는 declarator 를 해석한 후, 'start_function' 는 여기에 parameter names 의 list 혹은 decl 들의 chain 을 놓는다. 그리고 'store_parm_decls' 가 여기서 이러한 정보를 찾는다.

- current_function_parm_tags

위와 비슷하지만, last_function_parm_tags 용이다.

- last_function_parms

  function declarator 해석하면 여기에 parameter names 의 list 혹은 parameter decls 의 chain 를 남긴다.

- last_function_parm_tags

  function declarator 해석하면 여기에 parmlist 내 정의된 enum type 들과 structure 의 chain 을 남긴다.

그런후, TREE_STATIC (decl1) 에 1 을 넣음으로써, static storage 에 존재하도록 한다.

이제 Function name 이 정의된 decl 를 기록하게 되는데, pushdecl () 함수를 통해서 현재 binding level 에 기록하게 되는데, 만약 우리가 이 이름에 대한 decl 를 이미 가지고 있고, 그것이 FUNCTION_DECL 이라면, 이전 decl 를 사용하게 된다. 그리고 아래의 operation 을 통해서 새로운 binding level 을 만든다.

```
pushlevel (0);
declare_parm_level (1);
current_binding_level->subblocks_tag_transparent = 1;
```

위의 내용은 parameter 를 처리할 때, 앞에서 사용했던 구문인데, subblocks_tag_transparent 에 1 을 넣는 부분이 다르다.

이제 make_decl_rtl () 함수를 이용하여, FUNCTION_DECL 을 위한 DECL_RTL 을 생성하게 된다. 이 예제의 경우, 결과적으로 아래와 같은 RTX 가 생성되게 된다.

```
(mem:QI (symbol_ref:SI ("main")) [0 S1 A8])
```

이렇게 생성된 RTX 을 SET_DECL_RTL 매크로를 통해서 기록한다. 이제 반환값에 대한 node 를 만들어야 하는데, 아래와 같은 operation 을 통해서 만들어지게 된다.

```
restype = TREE_TYPE (TREE_TYPE (current_function_decl));
DECL_RESULT (current_function_decl)
  = build_decl (RESULT_DECL, NULL_TREE, restype);
```

이제, 전역 변수 immediate_size_expand 의 값을 복구한 후, start_fname_decls () 함수를 호출하고 start_function () 함수를 마치게 된다. 함수의 값이 완료된 후의 current_function_decl node 의 모습을 보면 아래와 같다.

```
<function_decl 0x401f6770 main
    type <function_type 0x401f6700
        type <integer_type 0x401da380 int SI
            size <integer_cst 0x401d7540 constant 32>
            unit size <integer_cst 0x401d75e0 constant 4>
            align 32 symtab 0 alias set -1 precision 32
            min <integer_cst 0x401d75a0 -2147483648>
            max <integer_cst 0x401d75c0 2147483647>
            pointer_to_this <pointer_type 0x401e2620>>
        DI
        size <integer_cst 0x401d7900 constant 64>
        unit size <integer_cst 0x401d7b20 constant 8>
        align 64 symtab 0 alias set -1
        arg-types <tree_list 0x401e5500 value <integer_type 0x401da380 int>
            chain <tree_list 0x401e5514 value <pointer_type 0x401f65b0>
                chain <tree_list 0x401e5528 value <void_type 0x401de770 void>>>>>
    public static QI file <stdin> line 2
```

```
result <result_decl 0x401f67e0 type <integer_type 0x401da380 int>
    SI file <stdin> line 2 size <integer_cst 0x401d7540 32>
    unit size <integer_cst 0x401d75e0 4>
    align 32> initial <error_mark 0x401dca20>
(mem:QI (symbol_ref:SI ("main")) [0 S1 A8])
chain <type_decl 0x401eed90 __g77_ulongint>>
```

6.3 함수의 중간

이 부분에서는, store_parm_decls () 함수에서 prototype 을 FUNCTION_DECL 에 적용을 한 후, 나머지 부분에서 발생하는 operation 에 대해 언급하도록 하겠다.

함수의 body 를 읽기 전에, 해놓는 일이 있는데, 그것은 전역 변수 cfun 를 초기화하고 현재 FUNCTION_DECL 를 적용하는 일과, statement tree 를 생성하는 일이다. 전자는 init_function_start () 함수를 통해서 이루어지고, 후자는 begin_stmt_tree () 함수를 통해서 이루어지게 된다.

우선 init_function_start () 함수에 대해서 살펴보도록 하자. 이 함수의 operation 은 아래와 같다.

1. prepare_function_start () 함수를 실행시켜, struct function 구조체를 초기화한다. 이에 대한 자세한 설명은 20 주 문서 "기반 작업 : (7) struct function 이란?" 를 참고하기 바란다.

2. 초기화가 완료가 되었기 때문에, 현재까지 획득한 정보를 struct function 에 기록한다. 기록되는 사항은 아래와 같다.

   - current_function_name

     현재 함수의 이름이 cfun→name 에 기록될 것이다.

   - cfun→decl

     현재 함수의 FUNCTION_DECL node 를 기록한다.

   - current_function_needs_context

     만약 이것이 static chain 을 사용하는 nested function 일 경우, 값이 0 이 아니다.

   - current_function_returns_struct
   - current_function_returns_pointer

   위 항목에 대한 각각에 대한 설명은 20 주 문서에 나와 있으므로 그것을 참고하기 바란다.

3. emit_line_note () 함수와 emit_note () 함수를 각각 실행하게 되는데, emit_line_note () 함수의 경우 함수의 첫번째 instruction 를 삭제하려는 시도를 막기 위해서 이며, 또한 final 에게 function prologue 전에 linenum 을 어떻게 output 할지를 말하기 위해서이다. emit_note () 함수가 실행되는 이유는 우리가 linenum 들을 원하지 않는다 하더라도, 첫번째 insn 는 note 임을 확실히 하기 위해서이다. 이것은 첫번째 insn 는 절대 지워지지 않도록 한다. 또한 final 는 거기에 note 가 나타날 거라 예상한다.

   > emit_line_note () 함수의 경우, 실행 과정이 emit_note 와 별반 다르지 않는데, cfun→stmt 에 emit_filename 와 emit_lineno 를 기록하는 부분이 다르며, 다른 부분에서는 모두 같다. emit_note () 함수의 경우 두번째 인자로 enum insn_note 를 넣을 수 있는데, 두번째 인자는 line number 로써, 0 보다 큰 값을 가지게 기본적으로 가지게 되지만, enum insn_note 가 같이 사용되기 위해서, insn_note 의 경우, 음수를 사용하여 구분하게 된다.
   >
   > `(note 2 0 0 NOTE\_INSN\_DELETED)`
   >
   > 위와 같은 note 가 만들어 진다고 가정을 하였을 때, emit_note () 함수에 의해서 생성된 rtx node 는 add_insn () 함수를 통해서, first_insn 와 last_insn 매크로를 통해, cfun→emit→x_first_insn 와 fun→emit→x_last_insn 에 각각 해당 rtx node 가 기록되게 된다.

   결과적으로는 cfun 의 내부를 설정하는 부분이 된다.

이제 init_function_start () 함수의 수행이 끝나면 선언된 함수를 위한 statement tree 를 아래의 구문을 실행함으로써, 시작하게 된다.

```
begin_stmt_tree (&DECL_SAVED_TREE (current_function_decl));
```

begin_stmt_tree () 함수의 경우, 간단한 operation 이 일어나는 함수로 아래의 내용이 전부라고 할 수 있다.

```
  /* 우리는 사소한 EXPR_STMT 를 생성하여서 last_tree 가 무엇이 따르건, 절대
     NULL 이 아니도록 한다.  우리는 finish_stmt_tree 내 관계없은 statement
     를 제거한다.  */
  *t = build_nt (EXPR_STMT, void_zero_node);
  last_tree = *t;
  last_expr_type = NULL_TREE;
  last_expr_filename = input_filename;
```

*t 부분의 tree node 는 실제 살펴볼 경우, 아래와 같은 모양을 갖게 된다.

```
<expr_stmt 0x401e5550
    arg 0 <integer_cst 0x401e0340 type <void_type 0x401de770 void> constant 0>>
```

결과적으로 DECL_SAVED_TREE (current_function_decl) 에 EXPR_STMT node 를 넣게 되는데, 이에 대한 node 정보를 각 세 개의 매크로 (위의 내용이 전역 변수처럼 보일지 모르지만, 엄연히 매크로이다.) 에 각각 알맞게 저장하게 된다. 아래의 하위섹션에서 statement list 에 대해서 알아 보도록 하자.

6.3.1 Statement Tree

Statement Tree 는 함수가 해석한 statement 에 대한 node 를 관리하는 list 이다. C 언어에서는 전역 구조체 변수 c_stmt_tree 가 이에 대한 정보를 가지고 있는데, 결과적으로 struct stmt_tree_s 를 살펴봄으로써, statement tree 를 알 수 있겠다.

Statement tree 에 관한 구조체는 $prefix/gcc/c-common.h 파일에 정의되어 있으며 아래와 같은 모양을 가지고 있다.

```
struct stmt_tree_s {
  tree x_last_stmt;
  tree x_last_expr_type;
  const char *x_last_expr_filename;
  int stmts_are_full_exprs_p;
};
```

각 구성 요소에 대해서 설명을 하면 아래와 같다.

- x_last_stmt

  Tree 에 추가된 마지막 statement.

- x_last_expr_type

  마지막 expression statement 의 type. (이 정보는 statement-expression extension 기능을 구현하는데 필요합니다.)

- x_last_expr_filename

  우리가 기록한 마지막 filename.

- stmts_are_full_exprs_p

C++ 에서 우리가, statement 를 full expression 으로 취급해야 할 경우 0 이 아닌 값을 가진다. 특히, 이 변수는 statement 의 끝 부분에서 우리가 해당 statement 동안 생성된 어떤 임시적인 것들을 제거해야 할 경우 0 이 아닌 값이다. 비슷하게, 만약 block 의 끝 부분에서, 우리가 이 block 내 어떤 local variable 들을 제거해야 할 경우가 존재할 수 있다. 보통, 이 변수는 0 이 아닌데, 그래서 이것은 C++ 의 normal semantic 인 경우가 많다.

하지만 tree structure 로써 aggregate initialization code 를 나타내기 위해서 우리는 statement expression 들을 사용한다. statement expression 내 statement 들은 전체 enclosing statement 가 완료될 때까지, cleanup 의 실행으로 마무리지어져서는 안된다.

이 flag 는 C 에서는 영향력을 가지지 않는다.

앞에서 begin_stmt_tree () 함수에서 나왔던 아래의 세개의 매크로에 대해서 살펴보면 아래와 같다.

- #define last_tree (current_stmt_tree ()→x_last_stmt)

  Statement-tree 를 생성할 때, 이것은 tree 에 추가된 마지막 statement 이다.

- #define last_expr_type (current_stmt_tree ()→x_last_expr_type)

  우리가 살펴본 마지막 expression-statement 의 type.

- #define last_expr_filename (current_stmt_tree ()→x_last_expr_filename)

  우리가 본 마지막 file 의 이름.

여기서 current_stmt_tree 에 대해 살펴보면, 전역 구조체 변수 c_stmt_tree 에 대한 주소를 반환하는 단순한 함수임을 확인할 수 있다.

6.4 함수의 끝 (finish_function)

함수는 Type, Declrator, Body 로 이루어진다는 것은 모두 알것이다. Body 를 해석하기 전에 GCC 에서는 시작과 끝을 구분하는데, 시작은 start_function () 함수이며, 끝은 finish_function () 함수로 마무리한다. 앞에서는 start_function () 함수에 대해서 이야기 하였다. 이제 이곳에서는 finish_function () 함수에 대해서 이야기 할것이다. 만약 여러 표현식이 함수의 body 에 선언되어 있을 경우, 복잡해져 설명하기 어려울 수 있으니, 위에서 언급했던 가장 간단한 함수 선언에 대해서 아래에서도 이를 중심으로 설명하겠다. 간단한 함수 선언이란 아래의 내용이다.

```
int
main (int argc, char *argv[]) {
}
```

위 예제의 body 에는 아무것도 선언된 것이 없기 때문에, start_function () 함수가 호출되자 마자 finish_function () 함수 또한 호출되게 된다. 우선 current_function_decl 에 대해 확실히 하기 위해 이에 대한 tree node 를 다시 한번 보고 가자. finish_function () 함수가 실행되었을 초기의 모습이다.

```
<function_decl 0x401f6770 main
    type <function_type 0x401f6700
        type <integer_type 0x401da380 int SI
            size <integer_cst 0x401d7540 constant 32>
            unit size <integer_cst 0x401d75e0 constant 4>
            align 32 symtab 0 alias set -1 precision 32
            min <integer_cst 0x401d75a0 -2147483648>
            max <integer_cst 0x401d75c0 2147483647>
            pointer_to_this <pointer_type 0x401e2620>>
        DI
        size <integer_cst 0x401d7900 constant 64>
        unit size <integer_cst 0x401d7b20 constant 8>
```

```
    align 64 symtab 0 alias set -1
    arg-types <tree_list 0x401e5500 value <integer_type 0x401da380 int>
        chain <tree_list 0x401e5514 value <pointer_type 0x401f65b0>
            chain <tree_list 0x401e5528 value <void_type 0x401de770 void>>>>>
public static QI file <stdin> line 1
arguments <parm_decl 0x401f6540 argc type <integer_type 0x401da380 int>
    SI file <stdin> line 1 size <integer_cst 0x401d7540 32>
    unit size <integer_cst 0x401d75e0 4>
    align 32 context <function_decl 0x401f6770 main>
    result <integer_type 0x401da380 int>
    initial <integer_type 0x401da380 int>
    arg-type <integer_type 0x401da380 int>
    arg-type-as-written <integer_type 0x401da380 int>
    chain <parm_decl 0x401f6690 argv type <pointer_type 0x401f65b0>
        unsigned SI file <stdin> line 1
        size <integer_cst 0x401d7b80 constant 32>
        unit size <integer_cst 0x401d7be0 constant 4>
        align 32 context <function_decl 0x401f6770 main>
        result <pointer_type 0x401f65b0>
        initial <pointer_type 0x401f65b0>
        arg-type <pointer_type 0x401f65b0>
        arg-type-as-written <pointer_type 0x401f65b0>>>
result <result_decl 0x401f67e0 type <integer_type 0x401da380 int>
    SI file <stdin> line 1 size <integer_cst 0x401d7540 32>
    unit size <integer_cst 0x401d75e0 4>
    align 32> initial <error_mark 0x401dca20>
(mem:QI (symbol_ref:SI ("main")) [0 S1 A8])
chain <type_decl 0x401eed90 __g77_ulongint>>
```

우선 간략한 finish_function () 함수의 실행 순서를 알아보도록 하자.

1. poplevel () 함수를 실행시켜서, store_parm_decls () 함수에서 실행한 pushlevel () 함수와 짝을 마춘다. poplevel () 함수의 실행을 살펴보자. 이 함수가 호출될 때, 인자로 keep 과 functionbody 가 각각 1 로 설정되어 호출되는데, 이로 인해 생기는 의미는 아래에서 조금씩 살펴볼 것이다. current_binding_level→names 내용을 살펴보면 아래와 같이 앞에서 인자를 선언할 때 정의하였던, PARM_DECL 들이 존재함을 확인할 수 있을 것이다.

```
<parm_decl 0x401f6540 argc
    type <integer_type 0x401da380 int SI
        size <integer_cst 0x401d7540 constant 32>
        unit size <integer_cst 0x401d75e0 constant 4>
        align 32 symtab 0 alias set -1 precision 32
        min <integer_cst 0x401d75a0 -2147483648>
        max <integer_cst 0x401d75c0 2147483647>
        pointer_to_this <pointer_type 0x401e2620>>
    SI file <stdin> line 1 size <integer_cst 0x401d7540 32>
    unit size <integer_cst 0x401d75e0 4>
    align 32 context <function_decl 0x401f6770 main>
    result <integer_type 0x401da380 int>
    initial <integer_type 0x401da380 int>
    arg-type <integer_type 0x401da380 int>
    arg-type-as-written <integer_type 0x401da380 int>
    chain <parm_decl 0x401f6690 argv>>
```

이런 의문이 들 수 있는데, 즉 (...) 가 호출되면서 poplevel 이 이미 호출되었는데, (...) 의 binding level 이 왜 function body 의 binding level 에 존재하느냐고 물을 수 있을 것이다. 이는 store_parm_decls () 함수가 실행되면서 해당 PARM_DECL 들을 function body 의 binding level 에 pushdecl 하게 되기 때문이다. 상식적으로 생각을 해보면, main 함수의 body 에서도 argc, argv 변수를 사용하기 때문에 function 의 body 에 이것들이 있는 것은 당연할 수 있다.

poplevel 의 인자인 keep 과 functionbody 가 1 이기 때문에 BLOCK tree node 를 생성하게 된다. 새로이 만들어진 BLOCK node 에는 BLOCK_VARS (block) 와 BLOCK_SUBBLOCKS (block) 가 각각 설정되게 된다. 현재 처리하고 있는 것이 function body 이기 때문에 clear_limbo_values () 함수를 실행하고, BLOCK_VARS (block) 를 0 으로 초기화한다. 왜냐하면 이것이 function 의 top level block 일 경우 Var 들은 function 의 parameter 들이며, 이것들은 FUNCTION_DECL 에서 대신 발견되기 때문에 굳이 BLOCK 에 남길 필요가 없기 때문이다. 이제 DECL_INITIAL (current_function_decl) 에 BLOCK node 를 저장을 하고 TREE_USED (block) 를 1 로 설정한 후, binding level 을 pop 한다.

2. DECL_INITIAL (fndecl) 와 DECL_RESULT (fndecl) 의 내용을 update 한다.
   BLOCK_SUPERCONTEXT (DECL_INITIAL (fndecl)) 와 DECL_CONTEXT (DECL_RESULT (fndecl)) 에 각각 현재 FUNCTION_DECL 을 기록하게 된다.

3. 만약 'setjmp' 가 이 fn 내에서 호출되었다면, 'register' declaration 을 따르도록 setjmp_protect_... () 함수를 실행한다.

4. finish_fname_decls () 함수를 실행한다.

5. 이 함수를 위한 statement tree 를 매듭짓기 위해 finish_stmt_tree () 함수를 실행한다. begin_stmt_tree 에서 추가된 fake extra statement 를 제거하고, last_tree 매크로를 이용하여 값을 NULL_TREE 로 변환한다.

6. 더 이상 필요없는 memory 를 깨끗히 하기 위해 free_after_parsing () 함수를 실행시킨다. Function 의 해석이 끝난 후 (아직 compile 되지는 않았음) garbage collection 이 memory 를 회수할 수 있도록, 안전하게 제거될 수 있는 cfun 내부의 state 관련 모든 부분들을 깨끗히 합니다.

7. free_after_compilation () 함수를 실행시킨다. 이 또한 cfun 내부의 여러 구조체를 free 하거나, NULL 값으로 변환한다.

8. 전역 변수 cfun 을 NULL 로 한다.

9. 현재 선언된 함수가 nested function 이 아닐 경우, 이 함수의 body 를 위한 RTL 을 생성하기 위해 c_expand_body () 함수를 실행한다. 실행이 완료된 후 error reporting routine 들이 우리가 function 밖에 있음을 알도록 하기 위해, 전역 변수 current_function_decl 를 NULL 로 설정한다. 그럼 c_expand_body () 함수에 대해서 간략히 보고 넘어가도록 하겠다. 이 함수의 경우, 각 FUNCTION_DECL node 가 완성된후, optimization 을 수행하고, 그에 대한 assembly output 를 내보는 것까지 포함하는 광범위한 크기이다. 이에 대한 설명은 이 강의가 끝마칠 때까지 이루어질 범위이기 때문에, 여기에서는 자세하게 언급하지 않을 것이다. 왜냐하면 앞으로도 계속 써나가야 할 범위이기 때문이다. 이 함수의 내에서는 다음과 같은 operation 이 일어나게 된다.

   (a) optimize_inline_calls () 함수를 수행하여, FN 의 body 내 inline function 들로의 호출을 expand 한다.

   (b) init_function_start () 함수를 수행하여, 함수를 위한 RTL code 를 초기화한다.

   (c) expand_function_start () 함수를 수행하여, 새 함수를 위한 RTL 를 시작하고, RTL 을 emit 하는데 사용되는 변수를 설정한다.

   (d) expand_main_function () 함수를 수행하여, 이 함수가 'main' 이면, global initializer 등 기타 등등을 실행시키기 위해 '__main' 로의 call 를 emit 한다.

   (e) expand_stmt () 함수를 수행하여 Statement T 와 그것의 substatement 들, 그것의 nesting level 에서의 다른 statement 들을 위한 RTL 을 생성한다.

(f) expand_function_end () 함수를 수행하여, 현재 함수의 끝 부분을 위한 RTL 을 생성한다.

(g) rest_of_compilation () 함수를 수행하여 현재 function 혹은 variable 에 대한 assembler code 를 output 한다. 이 함수는 각 top-level definition 이 해석된 후 (yyparse 내) inish_function 로부터 호출된다.

c_expand_body () 함수에 대한 설명은 이 정도에서 마무리 하도록 하겠다. 결과적으로는 FUNCTION_DECL node 에 대한 assembly code 를 내보낼 것이다. 이에 대해서는 다른 주에서 좀 더 자세하게 살펴볼 것이다.

제 7 절 Binding Level

아래에서는 위의 어떤 { ... } 나 (...), compound statement 와 같은 block 간에 구분할 때 사용될 필요가 있을 경우 사용하게 되는 binding level 에 대해서 알아보도록 하겠다.

7.1 Binding Level 구현을 위한 구조체

```
struct binding_level
  {
    tree names;
    tree tags;
    tree shadowed;
    tree blocks;
    tree this_block;

    struct binding_level *level_chain;

    char parm_flag;
    char tag_transparent;
    char subblocks_tag_transparent;
    char keep;
    char keep_if_subblocks;

    int n_incomplete;

    tree parm_order;
  };
```

각 binding_level 구조체를 이루는 구성 요소의 역활에 대해서 살펴보도록 하자.

- names

  모든 variable, constant, function, typedef type 들을 위한 _DECL 노드들의 chain. 이것들은 제공된 반대 순서로 있다.

- tags

  Tag 이름들을 찾기 위한 structure 와 union, enum 정의(definition)들의 리스트. TREE_LIST node 들의 chain 으로써, 각각의 것의 TREE_PURPOSE 는 이름이거나, NULL_TREE 이다; 또 각각의 TREE_VALUE 는 RECORD_TYPE 혹은 UNION_TYPE, ENUMERAL_TYPE node 이다.

- shadowed

각 level 을 위한 것으로, 이 level 이 pop 되어질 때 복구되어야 하는 shadowed outer-level local definition 들의 list 이다. 각 link 는 TREE_LIST 이며, 이것의 TREE_PURPOSE 는 identifier 이고, 이것의 TREE_VALUE 는 그것의 old definition (..._DECL node 종류인) 이다.

- blocks

  각 level 을 위한 것으로 (Global 인 것은 제외하고), 모든 level 에 대해 한 level 아래로 들어갔거나, 빠져나왔던 BLOCK node 들의 chain.

- this_block

  만약 미리 할당된 것이라면, 이 level 을 위한 BLOCK node 이다. 만약 0 일 경우, BLOCK 은 level 이 pop 되어질 때 (필요한 경우) 할당되어진다.

- level_chain

  이것이 포함되는 (어떤 것으로부터 내려오는) binding level.

- parm_flag

  0 이 아닐 경우 함수의 parameter 를 가지고 있는 레벨을 위해 사용됨.

- tag_transparent

  만약 이 레벨(level)이 tag 들을 위해 "존재하지 않을" 경우 0 이 아닌 값을 가집니다.

- subblocks_tag_transparent

  만약 이 level 의 sublevel 들이 tag 들이 "존재하지 않을 경우" 0 이 아니다. 이것은 function body 를 읽는 동안 함수 정의를 parm level 에서 설정된다. 그래서 function body 의 가장 외곽 block 은 tag-transparent 일 것이다.

- keep

  만약 값이 0 이 아니라면, 다른 것에 상관없이 이 level 을 위한 BLOCK 을 만들라는 의미이다.

- keep_if_subblocks

  0 이 아닐 경우 만약 이 레벨(level)이 어떤 하위블럭(subblock)들을 가지고 있다면 BLOCK 을 만듭니다.

- n_incomplete

  불완전한 structure 혹은 union type 들을 가지고 있는 'names' 내부의 decl 들의 갯수.

- parm_order

  Parmlist 내 어떤 forward-decl 들을 포함하지 않고 parm 들의 (예약된) specified order 를 주는 decl 들의 list. 이것으로 assign_parms 를 위한 적당한 order 를 parm 들에 놓을 수 있다.

7.2 Binding Level 을 위한 전역 변수

Binding Level 을 효율적으로 사용하기 위해서, GCC 에서는 $prefix/gcc/c-decl.c 파일에 binding level 과 관련하여 여러 전역 변수를 선언하여 사용한다.

```
static struct binding_level *current_binding_level;
```

현재 영향력을 있는 binding level.

```
static struct binding_level *free_binding_level;
```

재사용을 기다리는 binding_level 구조체의 chain.

```
static struct binding_level *global_binding_level;
```

File scope 의 이름들을 위한 가장 외각의 binding level. 이것은 컴파일러가 시작될 때 생성되며 실행되는 동안 계속 존재하여 영향을 미치게 된다.

```
static struct binding_level clear_binding_level
  = {NULL, NULL, NULL, NULL, NULL, NULL_BINDING_LEVEL, 0, 0, 0, 0, 0, 0,
     NULL};
```

Binding level 구조체들은 이것을 복사함으로써 초기화됩니다.

```
static int keep_next_level_flag;
```

값이 0 이 아닐 경우, 무조건 push 된 다음 level 을 위한 BLOC 을 생성함을 의미한다.

```
static int keep_next_if_subblocks;
```

값이 0 이 아닐 경우, 만약 push 된 다음 level 이 subblock 들을 가지고 있을 경우 BLOCK 을 만드는 것을 의미한다.

7.3 Binding Level 의 가장 외곽 level 의 초기화

실제로 binding level 을 C 언어를 해석하는 과정에서 사용하기 위해서, 위 하위 섹션에서 언급한 전역 변수를 초기화할 필요가 있는데, 초기화가 이루어지는 변수는 global_binding_level 와 current_binding_level 이다. 이 초기화가 이루어지는 부분은 실제로 Yacc parser 에 의해서 각각의 토큰이 해석되게 되는 시점이 아닌, “언어 의존적인 초기화”가 이루어지는 동안 수행되게 되며,$prefix/gcc/c-decl.c 파일에 선언되어 있는 c_init_decl_processing () 함수에 의해서 이루어지게 된다.

```
  pushlevel (0);
  global_binding_level = current_binding_level;
```

단순히 다음과 같은 operation 이 수행되면서 이루어진다고 할 수 있는데, 결과적으로는 global_binding_level 와 current_binding_level 가 같은 값을 가지게 된다.

7.4 pushlevel 과 poplevel 함수들

실제 Binding level 을 만드는 부분이 pushlevel () 함수에서 하는 일이며, 현재 적용되는 binding level 에서 빠져 나오는 부분을 poplevel () 함수가 하는 일인데, pushlevel () 함수의 경우, 단순히 struct binding_level 구조체를 할당하거나 재활용해서 기존의 binding level 를 위한 전역 변수의 값을 넣거나 초기화하는 수준의 일만한다. 즉 요약하면 아래와 같은 코드가 전부이다.

```
*newlevel = clear_binding_level;
newlevel->tag_transparent
  = (tag_transparent
     || (current_binding_level
         ? current_binding_level->subblocks_tag_transparent
         : 0));
newlevel->level_chain = current_binding_level;
current_binding_level = newlevel;
newlevel->keep = keep_next_level_flag;
keep_next_level_flag = 0;
newlevel->keep_if_subblocks = keep_next_if_subblocks;
keep_next_if_subblocks = 0;
```

하지만 poplevel () 함수의 경우, pushlevel 보다는 상대적으로 복잡한데, 해당 level 에서 pop off 하고, 이 level 에 들어오기 전에 영향력을 행사했었던 identifier-decl mapping 들의 상태를 복구해야 하며, poplevel () 함수가 호출될 때 입력되는 parameter 에 따라, 작동이 달라지게 된다. 각 parameter 에 대해 설명을 하면, KEEP 과 FUNCTIONBODY, REVER 가 존재한다.

만약 KEEP 이 0 이 아닐 경우, 이 level 은 explicit declaration 들을 가지고 있어서, 그것의 declaration 들과 symbol table output 을 위한 subblock 들을 기록하기 위해 해당 level 을 위한 "block" (BLOCK node) 를 생성한다.

만약 FUNCTIONBODY 가 0 이 아닌 값이라면 이 level 은 함수의 body 를 가르킵니다. 그래서 마치 KEEP 이 설정되어 있는 것처럼 block 을 생성합니다. 모든 label 이름들을 깨끗히 지웁니다.

만약 REVERSE 가 0 이 아닌 값이라면 그들을 BLOCK 속에 넣기 전에 decl 들의 순서를 반대로 합니다.

poplevel () 함수의 operation 에 대해서 나열하면 아래와 같은 순이다.

1. Decl 들이 쓰여진 순서대로 되어 있는 decl 들을 얻는다. 보통 current_binding_level→names 는 역순이지만, parameter decls 은 앞에서 forward order 로 놓았었다.
2. 이 block 내 어떤 nested inline function 들이 아직 ouput 되지 않았을 경우 이를 output 한다.
3. expand_end_bindings 내 RTL 을 생성하는 동안 사용되지 않은 변수에 대해 경고를 하였다. 하지만 function-at-a-time mode 에서는 전혀 function 으로 확장하기 않기 때문에, (예를 들면, auto inlining) 우리는 이것을 이제 분명히 한다.
4. 만약 해당 level 내에 어떤 declaration 들 혹은 structure tag 들이 존재하거나 만약 이 level 이 function body 이라면, 이 함수의 life 를 위해 그들을 기록하도록 BLOCK 을 생성한다.
5. 각 subblock 에다가, 이 block 이 superior 라고 기록한다.
6. 이 level 의 local variable 들의 의미를 깨끗히 한다.
7. 이 level 에 의해 shadow 되어졌단 outer level 들의 모든 name-meaning 들을 복구한다.
8. 만약 exit 되어지고 있는 level 이 함수의 top level 일 경우, 모든 level 들을 검사하고, 그들의 names 의 현재 (function local) meaning 들을 깨끗히 한다.
9. 현재 level 을 pop 하고, 재활용을 위해 structure 를 free 한다.
10. 몇몇 더 높은 level 들에 만들었던 block 들을 정리한다. 이제 막 Exit 된 level 을 위한 block 을 우리가 만들지 않았을 경우, Inner level 들을 위해 만들어졌던 어떤 block 들 (그래서 해당 level 의 subblock 들로 기록될 수 없기 때문에) 은 반드시 앞쪽으로 옮겨져야 하며 그래서 그들이 나중에 다른 것의 subblock 이 될 수 있도록 한다.

11. 이 binding contour 에 따르는, 모든 tagged type 들을 위한 TYPE_CONTEXT 들을 설정한다. 그래서 그들이 적당한 construct 를 가르키게 되는데, 즉 현재 FUNCTION_DECL node 를 가르킬 수도 있고, 이제막 건설된 BLOCK node 를 가르킬 수 있다.

    tagged types 의 scope 가 몇몇 function type specification 를 위한 formal parameter list 일 경우, 우리는 여기서 그들의 TYPE_CONTEXT 들을 적당히 설정할 수 없는데, 그것은 우리가 이용가능한 적당한 FUNCTION_TYPE node 로의 pointer 를 가지고 있지 않고 있기 때문임을 참고하라. 이러한 경우를 위해서 relevant tagged type node 들의 TYPE_CONTEXT 들은 이러한 "parameter list local" tagged types 들을 위한 "scope" 를 나타내는 FUNCTION_TYPE node 가 생성되자마자 'grokdeclarator' 에 의해서 설정이 될 것이다.