# 기반 작업
# (1) struct cpp_reader 란?

정원교

2004년 3월 17일

## 목 차



## 제 1 절 6 주째 강의를 시작하며

안녕하세요. 정원교입니다. 제가 현실감각이 둔해졌는지 모르겠지만 어느새 봄이 완연히 와 있더군요. 어제 오랜만에 낮에 일어나 (-_-; 저는 보통 아침 8 시에 자서 오후 5시쯤 일어남) 밖에 나가보니 제가 입고 있던 겨울 잠바가 저를 덥게 만들더군요. 제가 그동안 입고 있던 옷을 모두 장롱속에 넣어 버리고 봄 옷을 꺼내 입었습니다. 새로운 마음으로 강의를 쓰는 작업과 직장 일들을 할 수 있어 너무 좋습니다.

강의도 벌써 6 주째 접어들면서 GCC 에 대해서 전체적인 시각으로도 보고, 내부에서 사용되는 옵션이 어떤 것들이 있고 그것이 어떻게 설정되는지도 보았습니다. 그리고 lang_hook 에 대해서도 보았습니다. 이번 6 주에서는 CPP (C pre-precessor) 에서 기본이 되는 구조체 cpp_reader 에 대해서 알아보도록 하겠습니다.

## 제 2 절 struct cpp_reader 란?

struct cpp_reader 는 $prefix/gcc/cpphash.h 파일 내부에 선언되어 있습니다. cpp_reader 는 전처리기 수행 "상태"를 캡슐화하고 있습니다. 아래는 strcut cpp_reader 구조체의 모습입니다. 이 구조체 또한 많은 다른 하위 구조체들을 포함하고 있으며 이것의 기능에 대해서는 조금씩 알아 보도록 하겠습니다. 하지만 너무 많은 깊은 내용들을 담을려고 하지는 않겠습니다. : ) 사실 저도 잘 모른답니다. 자 봅시다!

```
struct cpp_reader
{
  cpp_buffer *buffer;

  struct lexer_state state;

  struct line_maps line_maps;
  const struct line_map *map;
  unsigned int line;

  unsigned int directive_line;

  _cpp_buff *a_buff;
  _cpp_buff *u_buff;
  _cpp_buff *free_buffs;

  struct cpp_context base_context;
  struct cpp_context *context;

  const struct directive *directive;

  const cpp_hashnode *mi_cmacro;
  const cpp_hashnode *mi_ind_cmacro;
  bool mi_valid;

  cpp_token *cur_token;
  tokenrun base_run, *cur_run;
  unsigned int lookaheads;

  unsigned int keep_tokens;

  unsigned int errors;

  unsigned int mls_line;
  unsigned int mls_col;

  unsigned char *macro_buffer;
  unsigned int macro_buffer_len;

  struct splay_tree_s *all_include_files;
```

```
  unsigned int max_include_len;

  cpp_token date;
  cpp_token time;

  cpp_token avoid_paste;
  cpp_token eof;

  struct deps *deps;

  struct obstack hash_ob;
  struct obstack buffer_ob;

  struct pragma_entry *pragmas;

  struct cpp_callbacks cb;

  struct ht *hash_table;

  struct cpp_options opts;

  struct spec_nodes spec_nodes;

  unsigned char print_version;
  unsigned char our_hashtable;
};
```

각 구성요소들의 역활에 대해서 알아 봅시다.

- `buffer`
  buffer stack 의 TOP.
- `state`
  Lexer 상태.
- `line_maps`
  `map, line` Source 줄번호 추적 관련.
- `directive_line`
  현재 directive '#' 의 줄번호.
- `a_buff`
  Aligned permanent storage.
- `u_buff`
  Unaligned permanent storage.
- `free_buffs`
  Free buffer chain.
- `base_context, context`
  Context stack 관련 부분
- `directive`
  만약 in_directive 라면, 인식한 directive 를 가짐.

- `mi_cmacro, mi_ind_cmacro, mi_valid`
  Multiple include optimisation.

- `cur_token`
  각 token 을 해석할 때 마다 현재 해석한 token 의 정보를 잡고 있는 cpp_token 구조체를 가지고 있다. 이를 이용하여 현재 GCC 가 무슨 token 을 해석하였는지 확인할 수 있다.

- `base_run, *cur_run`
  struct tokenrun 구조체는 GCC 가 읽어들이는 수많은 token 을 관리하는 구조체이다. base_run 구성요소는 전체적인 tokenrun 의 시작 부분을 가르키는 요소이다. 즉, 이 구성요소의 prev 나 next 가 생겨나더라도, 처음에 만들어 진 것이 이것이다라고 계속 가르키고 있다. cur_run 은 현재 활동중인 tokenrun 을 가르키고 있다. 대부분 250 개 정도의 token 을 위한 공간을 가지고 있으며, 이 공간이 꽉 다찼을 경우에, cur_run 은 새로운 값으로 변경되게 된다.

- `lookaheads`
  Lexing 관련 부분

- `keep_tokens`
  값이 0 이 아닐 경우 lexer 가 token 의 실행을 재사용하는 것으로 부터 막습니다.

- `errors`
  exit code 를 위한 오류 counter.

- `mls_line, mls_col`
  문자열 constant (multi-line 문자열들)에서 newline 이 처음 보였던 행과 열.

- `macro_buffer, macro_buffer_len`
  Macro 정의 문자열들을 잡고 있을 buffer.

- `all_include_files`
  다른 include 된 파일에 관한 TREE. cppfiles.c 파일을 보세요.

- `max_include_len`
  include 파일들을 위한 탐색 경로에서의 디렉토리 이름 중 현재 최대 길이. (우리가 얻은 것과 동일.)

- `date, time`
  날짜와 시간 token 들. 만약 그것들이 계산될 필요성이 있을 경우 계산됨.

- `avoid_paste, eof`
  EOF token 과 붙이기 강제 회피 (forcing paste avoidance) token.

- `deps`
  mkdeps.c 의 의존성들을 다루는 opaque. -M 옵션등에서 사용됨.

- `hash_ob`
  Obstack 은 모든 macro hash node 들을 가지고 있습니다. 이것은 결코 줄어들거나 수축하지 않습니다. cpphash.c 를 보십시오.

- `buffer_ob`
  buffer 와 conditional 구조체들을 가지고 있는 obstack. 이것은 실제 stack 입니다. cpplib.c 를 보세요.

- `pragmas`
  Pragma 테이블 - 라이브러리 사용자가 인식된 pragma 들에 대한 목록을 추가할 수 있기 때문에 동적임.

- `cb`
  Call back 들.

- `hash_table`
  식별자(identifier) hash table.
- `opts`
  사용자가 볼 수 있는 옵션들을 담고 있습니다. 이에 대한 자세한 설명은 아래에서 계속하게 됩니다.
- `spec_nodes`
  특별한 node 들 - 전처리기를 위해 미리 정의된 의미를 가지는
  식별자(identifier) 들을 위한 것.
- `print_version`
  우리의 version number 를 출력할지 안할지. 이렇게 함으로써 -v -version 들 때문에 두번 값을 얻지 않습니다.
- `our_hashtable`
  cpplib 가 hashtable 을 소유하고 있는지 없는지.

이 구조체는 처음 lang_hook.init_options () 함수에 의해 초기화되는데, C 언어의 경우 $prefix/gcc/c-common.c 파일에 선언되어 있는 cpp_create_reader () 함수에 의해서 초기화가 이루어 집니다. 이 함수에서 이루어지는 수행에 대해서는 다른 절에서 하도록 하겠습니다.

이 구조체는 다른 많은 구조체들을 내부에 포함하고 있습니다. 포함되는 구조체는 이 구조체내에서만 사용되는 것도 있고, 다른 곳에서도 사용되는 구조체들이 있습니다. 우선 이 하위 절로써 각각에 포함되는 구조체를 바탕으로 하여서 적어 나가도록 하겠습니다. 보게될 구조체의 순서는 아래와 같습니다.

1. struct cpp_options
2. cpp_buffer
3. struct lexer_state
4. struct line_maps
5. _cpp_buff
6. struct cpp_context
7. struct directive
8. cpp_hashnode
9. cpp_token
10. tokenrun
11. struct deps
12. struct pragma_entry
13. struct cpp_callbacks
14. struct ht
15. struct spec_nodes

휴 상당히 많군요. 그리고 이 내부에도 다른 구조체가 포함되어 있을 것이구요. 조금씩 보도록 하겠습니다. struct cpp_options 부터 시작하겠습니다.

## 2.1 struct cpp_options

이 구조체는 struct cpp_reader 내에 nest 되는 구조체입니다. 그리고 command line 에서 볼수 있는 많은 option 들을 지니고 있습니다. 구조체의 모습은 아래와 같습니다. 상당히 많은 량의 옵션을 가지고 있다는 사실을 아실 수 있습니다. 위에서와 마찬가지로 각 항목에 대한 설명을 아래에서 나열하도록 하겠습니다.

```
struct cpp_options
{
  const char *in_fname;
  const char *out_fname;

  unsigned int tabstop;

  struct cpp_pending *pending;

  const char *deps_file;

  struct search_path *quote_include;
  struct search_path *bracket_include;

  struct file_name_map_list *map_list;

  const char *include_prefix;
  unsigned int include_prefix_len;

  const char *user_label_prefix;

  enum c_lang lang;

  unsigned char verbose;
  unsigned char signed_char;
  unsigned char cplusplus;
  unsigned char cplusplus_comments;
  unsigned char objc;
  unsigned char discard_comments;
  unsigned char trigraphs;
  unsigned char digraphs;
  unsigned char extended_numbers;
  unsigned char print_deps;
  unsigned char deps_phony_targets;
  unsigned char print_deps_missing_files;
  unsigned char print_deps_append;
  unsigned char print_include_names;
  unsigned char pedantic_errors;
  unsigned char inhibit_warnings;
  unsigned char warn_system_headers;
  unsigned char inhibit_errors;
  unsigned char warn_comments;
  unsigned char warn_trigraphs;
  unsigned char warn_import;
  unsigned char warn_traditional;
  unsigned char warnings_are_errors;
  unsigned char no_output;
```

```
  unsigned char remap;
  unsigned char no_line_commands;
  unsigned char ignore_srcdir;
  unsigned char dollars_in_ident;
  unsigned char warn_undef;
  unsigned char c99;
  unsigned char pedantic;
  unsigned char preprocessed;
  unsigned char no_standard_includes;
  unsigned char no_standard_cplusplus_includes;
  unsigned char dump_macros;
  unsigned char dump_includes;
  unsigned char show_column;
  unsigned char operator_names;
  unsigned char help_only;
};
```

각 구성요소에 대한 자세한 설명은 아래와 같습니다.

- `in_fname, out_fname`
  input 과 output 파일들의 이름.
- `tabstop`
  Tab stop 사이의 문자수.
- `pending`
  Pending option 들 : -D, -U, -A, -I, -ixxx.
- `deps_file`
  deps 가 written 하고 있을 파일이름. 만약 deps 가 stdout 에 written 하고 있다면 값은 0 이 됨.
- `quote_include, bracket_include`
  include 파일들을 위한 찾기 경로들. 여기서 quote_include 는 “” 를 나타내고, bracket_include 는 <> 를 나타냅니다.
- `map_list`
  Header 이름들과 파일 이름들간의 map, 이것은 파일이름의 길이가 제한이 있는 DOS 상에서만 사용됩니다.
- `include_prefix, include_prefix_len`
  표준 include 파일 디렉토리들내에서
  ‘/usr/lib/gcc-lib/TARGET/VERSION’ 를 대체해야 하는 디렉토리 접두사.
- `user_label_prefix`
  -fleading_underscore 옵션이 이것을 “_” 로 설정.
- `lang`
  우리가 전처리하고 있는 언어.
- `verbose`
  0 이 아닐 경우 옵션 -v 가 설정되었음을 의미합니다. 그래서 include 디렉토리들의 모든 사항을 출력합니다.
- `signed_char`
  0 이 아닐 경우 char 는 sign 을 의미합니다.

- `cplusplus`
  0 이 아닐 경우 C++ 용 extra default include directory 들을 사용하는 것을 의미합니다.
- `cplusplus_comments`
  0 이 아닐 경우 C++ 스타일의 주석을 다루는 것을 의미합니다.
- `objc`
  0 이 아닐 경우 objective C 용 #import 를 다루는 것을 의미합니다.
- `discard_comments`
  0 이 아닐 경우 output 파일에 comment 들을 복사하지 않습니다.
- `trigraphs`
  0 이 아닐 경우 ISO trigraph sequence 들을 처리함을 의미.
- `digraphs`
  0 이 아닐 경우 ISO digraph sequence 들을 처리함을 의미.
- `extended_numbers`
  0 이 아닐 경우 hexadecimal float 들과 LL 접미사를 허용함을 의미.
- `print_deps`
  0 이 아닐 경우 전처리된 output 보다 include 된 파일들의 이름을 먼저 출력함을 의미합니다. 1 은 단순히 #include "..." 를 의미하고 2 는 #include <...> 를 의미합니다.
- `deps_phony_targets`
  0 이 아닐 경우 각 header 를 위한 엉터리(phony) target 들이 생성됩니다.
- `print_deps_missing_files`
  0 이 아닐 경우 -M output 내에서 .h 파일들이 빠져있을 경우 파일들이 생성되었고 오류는 없었다고 가정합니다.
- `print_deps_append`
  만약 참이면 fopen (deps_file, "a") 그렇지 않으면 fopen (deps_file, "w").
- `print_include_names`
  0 이 아닐 경우 header 파일들의 이름을 출력합니다. (-H)
- `pedantic_errors`
  0 이 아닐 경우 cpp_pedwarn 이 hard error 를 발생함을 의미.
- `inhibit_warnings`
  0 이 아닐 경우 경고 메세지들을 출력하지 않음을 의미.
- `warn_system_headers`
  0 이 아닐 경우 system header 들로부터의 경고를 숨기지 않음을 의미.
- `inhibit_errors`
  0 이 아닐 경우 오류 메세지들을 출력하지 않습니다. 그것을 선택하는 옵션은 없지만 cpplib 의 사용자에 의해서 설정되어 질 수 있습니다. (예를 들면, fix-header)
- `warn_comments`
  0 이 아닐 경우 comment 내부에 slash-start(/*) 가 나타난다면 경고함을 의미.
- `warn_trigraphs`
  0 이 아닐 경우 만약 어떠한 trigraph 가 존재한다면 경고함을 의미.
- `warn_import`
  0 이 아닐 경우 만약 #import 가 사용되면 경고하도록 합니다.

- `warn_traditional`
  0 이 아닐 경우 traditional C 와 비호환적인 여러 요소에 관해서 경고함을 의미.
- `warnings_are_errors`
  0 이 아닐 경우 경고를 오류로 변경합니다.
- `no_output`
  0 이 아닐 경우 output 을 내놓지 않습니다, 하지만 side effect 를 가지는 #define 과 같은 directive 들은 여전히 그대로 움직입니다.
- `remap`
  0 이 아닐 경우 우리는 파일 이름들을 remap 하기위해 header.gcc 파일들을 살펴보아야 합니다.
- `no_line_commands`
  0 이 아닐 경우 줄 번호(line number) 정보에 대한 output 을 하지 않습니다.
- `ignore_srcdir`
  0 이 아닐 경우 -I- 가 발견되었으며 그래서 source-file 디렉토리에서의 #include “foo” 를 살펴보지 않습니다.
- `dollars_in_ident`
  값이 0 이면 dollar 표시는 구두점(punctuation)임을 의미.
- `warn_undef`
  0 이 아닐 경우 #if 내에서 정의되지 않은 식별자가 다루어지는 것에 대해서 경고함을 의미.
- `c99`
  0 이 아닐 경우 corrigenda 와 amendments 를 포함한 1999 C Standard 를 가르킴.
- `pedantic`
  0 이 아닐 경우 ANSI 표준이 요구하는 모든 오류 메세지를 표시함을 의미.
- `preprocessed`
  0 이 아닐 경우 우리는 이미 전처리가 끝난 code 를 보고 있는 뜻으로 macro expansion 과 같은 것을 수행하는데 괴로워하지 않아도 된다는 의미입니다.
- `no_standard_includes`
  0 이 아닐 경우 header 들을 위한 모든 표준 디렉토리들을 disable 합니다.
- `no_standard_cplusplus_includes`
  0 이 아닐 경우 header 들을 위한 C++-specific 표준 디렉토리들을 disable 합니다.
- `dump_macros`
  0 이 아닐 경우 몇몇 유형의 macro 들을 dump 함 -¿ 위를 참조.
- `dump_includes`
  0 이 아닐 경우 #include 관련 행들을 output 으로 내보내버리고 통과합니다.
- `show_column`
  오류 메세지에 column 번호를 출력합니다.
- `operator_names`
  0 이 아닐 경우 C++ alternate operator name 들을 다루는 것을 의미합니다.
- `help_only`
  만약 option 에서 –help 혹은 –version, –target-help 가 보인다면 참입니다. Stand-alone CPP 는 옵션을 해석한 후 빠져나가야 합니다; driver 들은 help 의 출력을 계속하길 원할 것입니다.

이 구조체는 비록 구조체 struct cpp_reader 에 포함되는 것이지만 이 구조체 또한 다른 구조체를 포함하고 있습니다. 이 내부적인 것들에 대해서 광적으로 파고들어가 봅니다. (어느 누가 먼저 지치는가 봅시다. -_-;) 다음 페이지에서 만나요.

### 2.1.1 struct cpp_pending

먼저 구조체 struct cpp_options 에 포함되어 있는 여러 구조체 중에서 struct cpp_pending 구조체에 대해서 알아 봅시다. 이 구조체는 $prefix/gcc/cppinit.c 파일에 선언되어 있으며 구조체의 모습은 아래와 같습니다.

```
struct cpp_pending
{
  struct pending_option *directive_head, *directive_tail;

  struct search_path *quote_head, *quote_tail;
  struct search_path *brack_head, *brack_tail;
  struct search_path *systm_head, *systm_tail;
  struct search_path *after_head, *after_tail;

  struct pending_option *imacros_head, *imacros_tail;
  struct pending_option *include_head, *include_tail;
};
```

'pending' 구조체는 우리가 cpp_read_main_file 함수를 호출하기 전까지 실제로 처리되지 않는 모든 옵션들을 모아놓는 구조체입니다. 이것은 여러 list 들과 옵션의 각 type 을 위한 것들로 이루어져 있습니다. 그리고 quick insertion 을 위한 head 와 tail 포인터들을 가지고 있습니다.

불행하게도 이 구조체 마저 내부 다른 구조체를 가지고 있습니다. 여기에 포함되어 있는 구조체에 대한 것을 계속 추적해 나갑시다. 먼저 struct pend_option 을 봅시다. 그림 1 에 원형이 있습니다.

```
typedef void (* cl_directive_handler)
                          PARAMS ((cpp_reader *, const char *));
struct pending_option
{
  struct pending_option *next;
  const char *arg;
  cl_directive_handler handler;
};
```

그림 1: struct pending_option

'struct pending_option' 는 하나의 -D 혹은 -A, -U, -include, -imacros switch 를 기억하는 역활을 하는 구조체입니다. 이 구조체는 $prefix/gcc/cpphash.h 에 선언되어 있습니다.

다음으로 struct search_path 구조체에 대해 살펴보도록 합시다. 원형은 그림 2 을 보시면 됩니다.

이 구조체는 include 파일들을 살펴볼 디렉토리들의 목록을 포함하고 있습니다. 각 요소에 대해서 간단하게 언급합니다.

- `name, len`
  NAME 은 현재 파일의 디렉토리의 경우에 문자열이 null 로 끝나지 않을 수 있습니다!
- `ino, dev`
  여기서 언급된 디렉토리가 search path 상에서의 이전 디렉토리와 중복된다는 사실에 대해 말하는데 사용합니다.
- `sysp`
  0 이 아닐경우 이것은 system include 디렉토리입니다.
- `name_map`
  이 디렉토리를 위한 파일 이름들에 대한 mapping. 단지 MS-DOS 와 관계되는 플랫폼에서만 사용됩니다.

```
struct search_path
{
  struct search_path *next;

  const char *name;
  unsigned int len;
  ino_t ino;
  dev_t dev;
  int sysp;
  struct file_name_map *name_map;
};
```

그림 2: struct search_path

이 구조체 또한 struct file_name_map 라는 구조체를 포함하고 있군요. 추적은 계속됩니다.

struct file_name_map 구조체는 $prefix/gcc/cppfiles.c 파일에 선언되어 있습니다. 이에 대한 구조체 모습은 그림 3 에 나와 있습니다.

```
struct file_name_map
{
  struct file_name_map *map_next;
  char *map_from;
  char *map_to;
};
```

그림 3: struct file_name_map

file_name_map 구조체는 특정 디렉토리를 위한 파일 이름들의 mapping 을 가지고 있습니다. 이 mapping 은 그 디렉토리내에 FILE_NAME_MAP_FILE 이라고 명명된 파일로부터 읽습니다. 그러한 파일은 DOS 와 같이 파일 이름에 여러 제약을 가지고 있는 파일시스템 상에서 파일이름들을 map 하는데 사용될 수 있습니다. "파일 이름 map 파일"의 형식은 각 줄마다 두개의 token 을 가지는 여러 줄들로 구성됩니다. 처음 token 은 map 하기 위한 이름이며, 두번째 token 은 사용할 실제 이름입니다.

여기서 사용하는 FILE_NAME_MAP_FILE 파일은 GCC 내부에 이미 선언되어 있는데, 파일 이름은 "header.gcc" 를 사용하도록 합니다.

여기까지 분석함으로써 struct cpp_options 의 하위 구조체
struct cpp_pending 구조체에 대해서 알아 보았습니다. 이제 struct cpp_options 의 다른 하위 구조체에 대해 알아보도록 합시다. 그런데 알고보니 다른 하위 구조체에 속하는 struct search_path 구조체의 경우 struct cpp_pending 구조체를 알아보면서 한번 보았군요. 이 부분은 지나가기로 하고
struct file_name_map_list 구조체에 대해서 알아 봅시다.

이 구조체는 디렉토리당 하나씩 파일 이름 map 들에 관한 linked list 를 가지고 있습니다. 구조체에 대한 원본 모습은 그림 4 에 나와 있습니다. 이 구조체는 내부적으로 struct file_name_map 를 포함하고 있지

```
struct file_name_map_list
{
  struct file_name_map_list *map_list_next;
  char *map_list_name;
  struct file_name_map *map_list_map;
};
```

그림 4: struct file_name_map_list

만 이에 대해서는 위에 대해 언급하였으므로 더 이상 살펴보지 않겠습니다.
그리고 struct cpp_options 구조체 마지막으로

enum c_lang

에 대해서 알고 넘어 갑시다. 이 열거형은 C 언어의 종류에 대해서 나열하고 있는데, 아래와 같은 종류가 여기에 속하게 됩니다.

- CLK_GNUC89 = 0
- CLK_GNUC99
- CLK_STDC89
- CLK_STDC94
- CLK_STDC99
- CLK_GNUCXX
- CLK_CXX98
- CLK_OBJC
- CLK_OBJCXX
- CLK_AS

이것들은 cpp_reader_init 호출시 사용됩니다.
휴 여기까지 struct cpp_options 옵션에 포함되어 있는 모든 구조체에 대해 알아 보았습니다. 하지만 이 구조체도 struct cpp_reader 구조체에 포함되어 있는 하나의 구조체라를 것을 잊으시면 안될 듯합니다. 앞으로 가야 할 태산들이 너무 많습니다. 앞으로도 계속 struct cpp_options 구조체에 포함되어 있는 다른 하위 구조체들을 계속 살펴볼도록 하겠습니다. 이제 cpp_buffer 입니다.

## 2.2 cpp_buffer

이 구조체는 struct cpp_options 구조체가 선언된 윗부분에 자리잡고 있습니다. 이는 cpplib 가 읽고 있는 파일의 내용을 나타냅니다. 원형은 아래와 같은 모양을 갖습니다.

```
struct cpp_buffer
{
  const unsigned char *cur;
  const unsigned char *backup_to;
  const unsigned char *rlimit;
  const unsigned char *line_base;

  struct cpp_buffer *prev;

  const unsigned char *buf;

  struct include_file *inc;
  struct if_stack *if_stack;

  unsigned int col_adjust;
  unsigned char saved_flags;
  const unsigned char *last_Wtrigraphs;
  unsigned char warned_cplusplus_comments;
```

```
  unsigned char from_stage3;
  unsigned char search_cached;
  bool return_at_eof;

  struct search_path dir;
};
```

각 구성요소에 대해서 설명하도록 하겠습니다.

- `cur`
  현재 위치
- `backup_to`
  만약 peeked character 를 원하지 않는다면.
- `rlimit`
  유효한 data 의 끝
- `line_base`
  현재 행(줄)의 시작
- `prev`
  자신의 이전 구조체를 가르킵니다.
- `buf`
  전체 character buffer.
- `inc`
  include table 을 가르키는 포인터; 만약 이것이 파일 버퍼라면 NULL 이 아닌 값을 가짐. include_next 에 사용되며 control macro 들을 기록하는데 또한 사용됩니다.
- `if_stack`
  이 파일의 시작점에서의 if_stack 의 값.
  include file 내에서의 쌍이 맞지 않은 #endif (등등) 을 금지하는데 사용됩니다.
- `col_adjust`
  Whitespace 에서 tab 에 기인하여(owing) 정렬된 token 열(column) 위치.
- `saved_flags`
  PREV_WHITE 와/혹은 AVOID_LPASTE 를 포함합니다.
- `last_Wtrigraphs`
  Lexer 가 작동하는 방식 때문에, -Wtrigraphs 가 때때로 같은 trigraph 에 대해 두 번 경고할 수 있습니다. 이것을 그것을 막는데 도움을 줍니다.
- `warned_cplusplus_comments`
  만약 우리가 이 파일에서 C++ comment 들에 관해 이미 경고 메세지를 뿌렸다면 참(true) 값을 가집니다. 경고 메세지는 -pedantic 옵션이 활성화 되었거나 -Wtraditional 이 선언된 C89 extended mode 에서만 발생되면 각 파일당 한번 보여줍니다. (만약 그렇게 하지 않으면 아주 지저분하게 될것입니다.)
- `from_stage3`
  만약 우리가 trigraph 들과 escaped newline 들을 처리하지 않는다면 true 입니다. 전처리된 input 과 command line directive 들, _Pragma buffer 들에 대해선 참(true)입니다.
- `search_cached`
  값이 0 이 아닐 경우 “” include file 들에 대한 검색을 하는 디렉토리가 이미 계산되어 졌고 아래 “dir” 내에 저장되어 있음을 의미합니다.

- `return_at_eof`
  EOF 에서, 버퍼는 자동으로 pop 됩니다. 만약 RETURN_AT_EOF 가 참이면, CPP_EOF token 은 그 때 반환(return)됩니다. 그렇지 않는다면 enclosing buffer 로부터 다음 token 이 반환됩니다.

- `dir`
  이 buffer 의 파일에 대한 directory. 그것의 NAME member 는 할당되지 않았으므로 우리는 그것을 free 하는데 걱정할 필요가 없습니다.

이 구조체는 하위 구조체를 포함하고 있습니다. 그에 대해서 살펴보도록 하겠습니다. 알아보아야 할 구조체는 아래와 같습니다.

1. struct include_file
2. struct if_stack
3. struct search_path

   이 구조체는 위에서 알아 보았으므로 다시 다루지 않겠습니다.

### 2.2.1 struct include_file

먼저 struct include_file 구조체입니다. 이 구조체는 $prefix/gcc/cppfiles.c 파일에 선언되어 있습니다. 원형은 아래와 같으며 이 구조체는 모든 include 들의 table 을 갖고 있는데 사용됩니다.

```
struct include_file
{
  const char *name;
  const cpp_hashnode *cmacro;
  const struct search_path *foundhere;
  const unsigned char *buffer;
  struct stat st;
  int fd;
  int err_no;
  unsigned short include_count;
  unsigned short refcnt;
  unsigned char mapped;
};
```

각 구성요소에 대해서 설명을 하도록 하겠습니다.

- `name`
  파일의 실제 경로 이름

- `cmacro`
  Reinclusion 을 막는 어떤 macro.

- `foundhere`
  #include_next 와 sysp 를 위한 파일이 발견된 search path 에서의 위치.

- `buffer`
  Cached 된 파일 content 로의 포인터

- `st`
  파일을 위한 stat(2) data 의 복사본

- `fd`
  파일을 열때 사용한 fd (잠시 저장만 함)

- `err_no`
  만약 파일을 여는데 실패했다면 그에 대한 errno
- `include_count`
  파일이 읽힌 횟수
- `refcnt`
  이 파일을 사용하는 stacked buffer 들의 갯수
- `mapped`
  file buffer 가 mmapp 되었음을 나타냄.

이 구조체 또한 다른 하위 구조체를 포함하고 있는데, struct search_path 는 위해서 설명하였으며, cpp_hashnode 구조체는 아래의 절에서 따로 보도록 하겠습니다.

### 2.2.2 struct if_stack

이제 struct if_stack 에 대해서 알아 보도록 하겠습니다. 이 구조체가 선언되어 있는 부분은 $prefix/gcc/cpplib.c 파일에 선언되어 있습니다. 현재 처리 과정에서의 조건부에 대한 stack 이며 성공 처리, 실패 처리한 것들을 모두 포함하고 있습니다. 원형은 아래와 같습니다.

```
struct if_stack
{
  struct if_stack *next;
  unsigned int line;
  const cpp_hashnode *mi_cmacro;
  bool skip_elses;
  bool was_skipping;
  int type;
};
```

각 구성요소에 대해서 알아 봅시다.

- `next`
  다음을 가르킵니다.
- `line`
  조건이 시작된 줄(행).
- `mi_cmacro`
  전체 파일 주위의 #ifndef 에 대한 macro
- `skip_elses`
  앞으로 나올 #else / #elif 를 무시 해도 될까요?
- `was_skipping`
  만약 entry 가 skip 중이라면.
- `type`
  가장 최근 조건부, (diagnostics 용).

여기에서도 선언된 cpp_hashnode 구조체는 아래의 절에서 따로 보도록 합시다.

## 2.3 struct lexer_state

이제 struct lexer_state 에 대해서 알아 봅시다. 이 구조체는
$prefix/gcc/cpphash.h 에 선언되어 있습니다.

```
struct lexer_state
{
  unsigned char in_directive;
  unsigned char skipping;
  unsigned char angled_headers;
  unsigned char save_comments;
  unsigned char lexing_comment;
  unsigned char va_args_ok;
  unsigned char poisoned_ok;
  unsigned char prevent_expansion;
  unsigned char parsing_args;
};
```

각 구성요소에 대해서 알아 봅시다.

- `in_directive`
  만약 이 값이 0 이 아닌 경우, 현재 줄(행)에서의 처음 token 이 CPP_HASH 인것을 나타낸다. 즉 # 문자가 그 줄 처음으로 나왔다는 소리이다. 이럴 경우 directive 로 그 문장이 꾸며진다는 이야기가 된다.

- `skipping`
  만약 우리가 실패한 conditional group 을 건너뛴다면 참(true)이다.

- `angled_headers`
  0 이 아닐 경우 각괄호(¡)로 둘러싸인 헤더들을 가지는 directive 내에 있음을 의미한다.

- `save_comments`
  0 이 아닐 경우 comment 들을 저장합니다. 만약 discard_comments 가 선언되어 있거나 #define 부분의 모든 directive 내에 있을 경우에는 비활성화됩니다.

- `lexing_comment`
  0 이 아닐 경우 우리는 mid-comment 상태이다.

- `va_args_ok`
  0 이 아닐 경우 __VA_ARGS__ 를 lexing 하는 것은 유효하다.

- `poisoned_ok`
  0 이 아닐 경우 poisoned identifier 들을 lexing 하는 것은 유효하다.

- `prevent_expansion`
  0 이 아닐 경우 macro expansion 을 막습니다.

- `parsing_args`
  0 이 아닐 경우 function-like macro 로 인자들을 해석합니다.

계속 추적해 나가겠습니다.

## 2.4 struct line_maps

struct line_maps 구조체를 봅시다. 이 구조체는 $prefix/gcc/line-map.h 파일에 선언되어 있습니다. 원형은 아래와 같습니다.

```
struct line_maps
{
  struct line_map *maps;
  unsigned int allocated;
  unsigned int used;

  int last_listed;
  unsigned int depth;
  bool trace_includes;
};
```

이 구조체는 연대기적 순으로의 line_map 구조체들의 집합들을 가지게 됩니다. 각 구성요소에 대한 설명을 하겠습니다.

- `maps`
  아직 정확한 설명이 없습니다. 추후 UPDATE 하겠습니다.
- `allocated`
  아직 정확한 설명이 없습니다. 추후 UPDATE 하겠습니다.
- `used`
  아직 정확한 설명이 없습니다. 추후 UPDATE 하겠습니다.
- `last_listed`
  제일 위의 including file 이 LAST_LISTED 로 시작하는 것과 상관없이 가장 최근에 목록화된 include stack. -1 은 아직 목록화된 것이 아무것도 없음을 가르킵니다.
- `depth`
  현재 파일을 포함하여 include stack 의 depth.
- `trace_includes`
  만약 참이면 include trace 를 출력합니다.

### 2.4.1 struct line_map

내부에 포함되어 있는 struct line_map 을 다시 살펴 봅시다. 이 구조체는 struct line_maps 구조체 바로 위에 선언되어 있으므로 쉽게 찾으실 수 있을 것입니다.

```
struct line_map
{
  const char *to_file;
  unsigned int to_line;
  unsigned int from_line;
  int included_from;
  ENUM_BITFIELD (lc_reason) reason : CHAR_BIT;
  unsigned char sysp;
};
```

각 항목에 대한 설명을 보겠습니다.

- `to_file`
  파일 이름을 가르킵니다.
- `to_line` 파일의 행을 가르킵니다. 이는 to_file 과 물리적 줄번호를 구성합니다.
- `from_line` 논리행 FROM_LINE 는 물리적 원시 파일 TO_FILE 의 행 TO_LINE 으로 mapping 합니다. 그리고 그 뒤로 집합 내에 다음 line_map 구조체까지 1 대 1 대응으로 map 합니다.

- `included_from`
  INCLUDED_FROM 는 현재 포함된 전체에서의 line mapping 과 관련된, 집합 내부에서의 index 입니다. include stack 의 맨 밑의 파일(들)은 이 집합을 -1 로 가집니다.
- `reason`
  REASON 은 이 line map 의 생성 이유를 가지고 있습니다.
- `sysp` SYSP 는 시스템 헤드용일 경우 1 을, C++ 에서 extern "C" 가 보호되어질 필요가 있을 경우 C 시스템 헤더 파일용으로 2 를 가집니다. 그렇지 않을 경우 0 을 가집니다.

### 2.4.2 enum lc_reason

이 구조체를 이루고 있는 구성 요소 중 reason 에 대해서 좀 더 살펴 보면, 이는 열거형으로 구성되어 있는데, 설명과 같이 "line map 의 생성 이유"를 가지고 있습니다. 그 구성요소는 간단하게 아래와 같습니다.

```
enum lc_reason {LC_ENTER = 0, LC_LEAVE, LC_RENAME};
```

이것은 add_line_map () 을 사용하여 줄 번호 변경을 추가하는데 따른 이유를 가르키고 있으며,

- LC_ENTER 는 새로운 파일을 포함할 때 사용합니다. 예를 들면 C 에서의 #include directive 의 경우.
- LC_LEAVE 는 파일의 끝에 도달하였을 때.
- LC_RENAME 은 파일 이름 혹은 줄 번호가 위의 이유와 다른 이유로 변경되었을 경우. ( 예를 들면 C 에서의 #line driective )

와 같은 이유를 포함하고 있습니다.

## 2.5 _cpp_buff

이 구조체는 $prefix/gcc/cpphash.h 에 선언되어 있습니다. 원형은 아래와 같습니다.

```
typedef struct _cpp_buff _cpp_buff;
struct _cpp_buff
{
  struct _cpp_buff *next;
  unsigned char *base, *cur, *limit;
};
```

이는 그냥 일반적인 메모리 buffer 를 나타냅니다.

## 2.6 struct cpp_context

struct cpp_context 에 대해서 알아보도록 하겠습니다. 이 구조체 또한 $prefix/gcc/cpphash.h 에 선언되어 있으며 아래와 같은 형태입니다.

```
typedef struct cpp_context cpp_context;
struct cpp_context
{
  cpp_context *next, *prev;

  union utoken first;
  union utoken last;

  _cpp_buff *buff;
```

```
  cpp_hashnode *macro;

  bool direct_p;
};
```

각 구성요소에 대해서 설명을 하도록 하겠습니다.

- `next, prev`
  struct cpp_context 구조체는 이중 연결 리스트로 되어 있습니다.
- `first, last`
  Base context 가 연속된 token 들과는 다른 context 들. 예를 들면, macro expansion 들과 expanded argument token 들.
- `buff`
  만약 NULL 이 아니라면 이 buffer 는 현재 context 에 관련하는 storage 를 위해 사용됩니다. context 가 pop 될 때 buffer 는 풀리게 됩니다.
- `macro`
  macro context 를 위한 macro node 입니다. 필요하지 않다면 NULL.
- `direct_p`
  만약 utoken element 가 token 이라면 true 이고 그렇지 않으면 ptoken 이다.

### 2.6.1 union utoken

이 구조체 내부적으로 아직 소개되지 않은 구조체가 존재합니다. 계속 추적해 나갑니다. union utoken 을 살펴봅시다. 이 구조체는 struct cpp_context 구조체 근처에서 발견하실 수 있습니다. 원형은 아래와 같습니다.

```
union utoken
{
  const cpp_token *token;
  const cpp_token **ptoken;
};
```

## 2.7 struct directive

이 구조체는 $prefix/gcc/cpplib.c 파일에 선언되어 있으며 하나의 #-directive 를 정의합니다. 그리고 그것을 어떻게 다룰 것인지에 대한 내용도 포함합니다. 함수의 원형은 아래와 같습니다.

```
typedef void (*directive_handler) PARAMS ((cpp_reader *));
typedef struct directive directive;
struct directive
{
  directive_handler handler;
  const U_CHAR *name;
  unsigned short length;
  unsigned char origin;
  unsigned char flags;
};
```

각 구성요소에 대한 설명은 아래와 같습니다.

- `handler`
  Directive 를 다룰 함수.

- `name`
  Directive 의 이름.
- `length`
  이름의 길이.
- `origin`
  Directive 의 기원.
- `flags`
  이 Directive 를 표현하는 기호.

이 구조체를 보면 origin field 가 존재하는데, 각각에 대해서 설명을 하도록 하겠습니다. origin field 에 존재할 수 있는 값은 총 세가지가 있습니다. 아래와 같으며 설명도 하겠습니다.

1. KANDR

   KANDR directive 들은 traditional (K&R) C 에서 유래되었습니다.

2. STDC89

   STDC89 directive 들은 1989 C standard 에서 유래되었습니다.

3. EXTENSION

   EXTENSION directive 들은 확장판입니다.

그리고 flags field 에는 총 네가지의 경우가 존재할 수 있는데, 아래에서 설명하도록 하겠습니다.

1. COND

   COND 는 조건부를 가르킵니다

2. IF_COND

   opening conditional 를 의미.

3. INCL

   INCL 는 "..." 와 <...> 를 각각 q-char 와 h-char 로 취급함을 의미합니다.

4. IN_I

   IN_I 는 이 directive 가 -fpreprocessed 에 영향을 받더라고 다루어져야 함을 의미합니다.
   (Callback hook 들을 가지는 directive 들이 존재합니다.)

## 2.8 cpp_hashnode

cpp_hashnode 구조체를 살펴보도록 하겠습니다. 이 구조체는
$prefix/gcc/cpplib.h 에 선언되어 있습니다.

```
struct cpp_hashnode
{
  struct ht_identifier ident;
  unsigned short arg_index;
  unsigned char directive_index;
  unsigned char rid_code;
  ENUM_BITFIELD(node_type) type : 8;
  unsigned char flags;
```

```
  union
  {
    cpp_macro *macro;
    struct answer *answers;
    enum cpp_ttype operator;
    enum builtin_type builtin;
  } value;
};
```

이 구조체는 모든 3 개의 C front end 들 사이에 공유되는 identifier node 의 공통 부분입니다. 또한 문법적 의미에서 identifier 의 superset 인 CPP identifier 들을 저장하는데 사용됩니다. 각 구성요소의 설명입니다.

- `ident`
- `arg_index`
  Macro argument index.
- `directive_index`
  Directive table 에서의 index.
- `rid_code`
  Rid code - 전처리기용.
- `type`
  CPP node type.
- `flags`
  CPP flag 들.
- `value`
  - `macro`
    만약 매크로라면..
  - `answers`
    Assertion 에 대한 대답들.
  - `operator`
    명명된 operator 를 위한 code.
  - `builtin`
    Builtin macro 를 위한 code.

이 구조체에도 여러 구조체들이 포함되어 있는데, 그 중 먼저 struct ht_identifier 를 살펴 보도록 하겠습니다. 이 구조체는 $prefix/gcc/hashtable.h 에 선언되어 있는 구조체로써 아래와 같은 원형을 갖습니다.

```
typedef struct ht_identifier ht_identifier;
struct ht_identifier
{
  unsigned int len;
  const unsigned char *str;
};
```

이것은 각 hash table entry 가 가르키고 있는 것이 무엇인지를 가르켜줍니다. 이것은 다른 object 속의 깊숙한 곳에 속하게 됩니다.

또 멤버를 이루는 ENUM_BITFIELD(node_type) type 에 대해서도 알아 보겠습니다. node_type 은 현재 세 가지 요소를 가지고 있는데, Hash node 의 각기 다른 취향들을 나타냅니다. 구성요소에 대한 나열과 설명은 아래와 같습니다.

1. NT_VOID

   "아직 정의되지 않음"을 나타냄

2. NT_MACRO

   몇몇 form 의 매크로.

3. NT_ASSERTION

   #assert 용 술어.

그리고 struct answer 구조체에 대해서도 알아 봅시다. 이 구조체는 $prefix/gcc/cpplib.c 에 선언되어 있습니다. 원형은 아래와 같습니다.

```
struct answer
{
  struct answer *next;
  unsigned int count;
  cpp_token first[1];
};
```

위에서 설명했듯이 Assertion 에 대한 대답에 대한 연결 리스트를 가지고 있습니다. 여기서 포함하고 있는 cpp_token 구조체는 아래에서 설명하겠습니다.

이제 cpp_macro 구조체에 대해서 알아 봅시다. 이 구조체는
$prefix/gcc/cppmacro.c 파일에 선언되어 있으며 원형은 아래와 같습니다.

```
struct cpp_macro
{
  cpp_hashnode **params;
  cpp_token *expansion;
  unsigned int line;
  unsigned int count;
  unsigned short paramc;
  unsigned int fun_like : 1;
  unsigned int variadic : 1;
  unsigned int syshdr   : 1;
};
```

각 구성요소에 대한 설명을 하겠습니다.

- `params`
  어떤것에 대한 parameter 들.
- `expansion`
  Replacement list 의 처음 token.
- `line`
  시작 행(줄) 번호.
- `count`
  Expansion 에서의 token 들의 갯수.
- `paramc`
  Parameter 들의 갯수.
- `fun_link`
  만약 function-like macro 라면 설정.

- `variadic`
  만약 variadic macro 라면 설정.

- `syshdr`
  만약 system header 에 정의된 macro 라면 설정.

이 구조체에 포함되어 있는 다른 구조체에 대한 설명은 위에서 하였으므로 다시 언급하지 않겠습니다. 또 살펴볼 열거형이 두개가 있는데,

```
enum cpp_ttype operator;
enum builtin_type builtin;
```

가 그것입니다. 이에 대해서 이제 알아보도록 하겠습니다. 이제 알아볼 enum cpp_ttype 의 경우 아주 많은 열거형들을 포함하고 있는데, 우선 선언이 어떻게 되어 있는지 살펴 봅시다.

```
#define TTYPE_TABLE                             \
  OP(CPP_EQ = 0,        "=")                    \
  OP(CPP_NOT,           "!")                    \
  OP(CPP_GREATER,       ">")    /* 비교 */      \
  OP(CPP_LESS,          "<")                    \
  OP(CPP_PLUS,          "+")    /* 연산 */      \
  OP(CPP_MINUS,         "-")                    \
  OP(CPP_MULT,          "*")                    \
  OP(CPP_DIV,           "/")                    \
  OP(CPP_MOD,           "%")                    \
  OP(CPP_AND,           "&")    /* bit 관련 */  \
  OP(CPP_OR,            "|")                    \
  OP(CPP_XOR,           "^")                    \
  OP(CPP_RSHIFT,        ">>")                   \
  OP(CPP_LSHIFT,        "<<")                   \
  OP(CPP_MIN,           "<?")   /* 확장 */      \
  OP(CPP_MAX,           ">?")                   \
\
  OP(CPP_COMPL,         "~")                    \
  OP(CPP_AND_AND,       "&&")   /* 논리 */      \
  OP(CPP_OR_OR,         "||")                   \
  OP(CPP_QUERY,         "?")                    \
  OP(CPP_COLON,         ":")                    \
  OP(CPP_COMMA,         ",")    /* 그룹화 */    \
  OP(CPP_OPEN_PAREN,    "(")                    \
  OP(CPP_CLOSE_PAREN,   ")")                    \
  OP(CPP_EQ_EQ,         "==")   /* 비교 */      \
  OP(CPP_NOT_EQ,        "!=")                   \
  OP(CPP_GREATER_EQ,    ">=")                   \
  OP(CPP_LESS_EQ,       "<=")                   \
\
  OP(CPP_PLUS_EQ,       "+=")   /* 연산 */      \
  OP(CPP_MINUS_EQ,      "-=")                   \
  OP(CPP_MULT_EQ,       "*=")                   \
  OP(CPP_DIV_EQ,        "/=")                   \
  OP(CPP_MOD_EQ,        "%=")                   \
  OP(CPP_AND_EQ,        "&=")   /* bit 관련 */  \
  OP(CPP_OR_EQ,         "|=")                   \
  OP(CPP_XOR_EQ,        "^=")                   \
  OP(CPP_RSHIFT_EQ,     ">>=")                  \
  OP(CPP_LSHIFT_EQ,     "<<=")                  \
  OP(CPP_MIN_EQ,        "<?=")  /* 확장 */      \
  OP(CPP_MAX_EQ,        ">?=")                  \
```

```
  /* CPP_FIRST_DIGRAPH 로 시작하는 digraph 들.  */      \
  OP(CPP_HASH,          "#")    /* digraph 들 */        \
  OP(CPP_PASTE,         "##")                   \
  OP(CPP_OPEN_SQUARE,   "[")                    \
  OP(CPP_CLOSE_SQUARE,  "]")                    \
  OP(CPP_OPEN_BRACE,    "{")                    \
  OP(CPP_CLOSE_BRACE,   "}")                    \
  /* 구두점에 대한 remainder.  순서는 상관없음.  */     \
  OP(CPP_SEMICOLON,     ";")    /* 구조체 */    \
  OP(CPP_ELLIPSIS,      "...")                  \
  OP(CPP_PLUS_PLUS,     "++")   /* 증가 */      \
  OP(CPP_MINUS_MINUS,   "--")                   \
  OP(CPP_DEREF,         "->")   /* 접근자 */    \
  OP(CPP_DOT,           ".")                    \
  OP(CPP_SCOPE,         "::")                   \
  OP(CPP_DEREF_STAR,    "->*")                  \
  OP(CPP_DOT_STAR,      ".*")                   \
  OP(CPP_ATSIGN,        "@")  /* Object C 에서 사용 */ \
\
  TK(CPP_NAME,          SPELL_IDENT)    /* 단어 */                    \
  TK(CPP_NUMBER,        SPELL_NUMBER)   /* 34_be+ta  */               \
\
  TK(CPP_CHAR,          SPELL_STRING)   /* 'char' */                  \
  TK(CPP_WCHAR,         SPELL_STRING)   /* L'char' */                 \
  TK(CPP_OTHER,         SPELL_CHAR)     /* stray 구두점 */            \
\
  TK(CPP_STRING,        SPELL_STRING)   /* "string" */                \
  TK(CPP_WSTRING,       SPELL_STRING)   /* L"string" */               \
  TK(CPP_HEADER_NAME,   SPELL_STRING)   /* #include 에서의 <stdio.h>    \
\
  TK(CPP_COMMENT,       SPELL_NUMBER) /* Comment output 시만.  */       \
                                      /* SPELL_NUMBER 는 DTRT 를 유발 */\
  TK(CPP_MACRO_ARG,     SPELL_NONE)     /* Macro 인자.  */              \
  TK(CPP_PADDING,       SPELL_NONE)     /* cpp0 용 whitespace.  */      \
  TK(CPP_EOF,           SPELL_NONE)     /* 파일 혹은 줄의 끝.  */

#define OP(e, s) e,
#define TK(e, s) e,
enum cpp_ttype
{
  TTYPE_TABLE
  N_TTYPES
};
#undef OP
#undef TK
```

'=' 로 구분되는 처음 두 그룹들은 전처리기 표현식들에서 볼 수 있습니다. 이것은 _cpp_parse_expr 내에서의 lookup table 관련 수행을 허락합니다.

CPP_LAST_EQ 까지의 처음 그룹은 '=' 뒤에 바로 올 수 있습니다. Lexer 는 ">>=" 와 같이 끝에 '=' operator 를 필요로 합니다. 이것은 ">>" 와 같이 '=' 로 끝나지 않는 것들과 같은 순서로 시작되기 때문에 짝을 찾기 위해서 입니다.

다른 열거형 enum builtin_type 에 대해서 알아 봅시다. 이 열거형은 Builtin macro 의 다른 취향들을 나타내는데, _Pragma 는 operator 이지만 효율적인 이유로 builitin code 와 함께 그것을 다룹니다. 원형의 모습은 아래와 같습니다.

```
enum builtin_type
{
```

```
  BT_SPECLINE = 0,              /* '__LINE__' */
  BT_DATE,                      /* '__DATE__' */
  BT_FILE,                      /* '__FILE__' */
  BT_BASE_FILE,                 /* '__BASE_FILE__' */
  BT_INCLUDE_LEVEL,             /* '__INCLUDE_LEVEL__' */
  BT_TIME,                      /* '__TIME__' */
  BT_STDC,                      /* '__STDC__' */
  BT_PRAGMA                     /* '_Pragma' operator */
};
```

## 2.9 cpp_token

이제 cpp_token 구조체입니다. 이 구조체는 $prefix/gcc/cpplib.h 에 선언되어 있습니다. 원형은 아래와 같습니다.

이 구조체는 전처리 token 에 대한 정보를 모두 가지게 된다. 이 구조체는 주의깊게 뭉쳐 놓아(packed)야 하는데, 32-비트 host 에서는 16 바이트, 64-비트 host 에서는 정확히 24 바이트를 차지해야 한다. 이 구조체는 아래와 같으며, GCC 가 해석한 모든 token 의 내용은 아래에 들어가게 된다.

```
struct cpp_token
{
  unsigned int line;
  unsigned short col;
  ENUM_BITFIELD(cpp_ttype) type : CHAR_BIT;
  unsigned char flags;

  union
  {
    cpp_hashnode *node;
    const cpp_token *source;
    struct cpp_string str;
    unsigned int arg_no;
    unsigned char c;
  } val;
};
```

각 구성요소에 대해 아래에 설명했다.

- unsigned int line;

  Token 의 첫 문자의 논리적 행 번호.

- unsigned short col;

  Token 의 첫 문자의 열 번호.

- ENUM_BITFIELD(cpp_ttype) type : CHAR_BIT;

  token type

- unsigned char flags;

  flag 들에 대한 정보를 가지고 있다. 이 flag 들은 cpp_token 을 위해서 존재하는데, 현재 총 7 가지의 flag 를 가지고 있으며, 각각의 아래와 같이 구성된다.

```
#define PREV_WHITE      (1 << 0)
#define DIGRAPH         (1 << 1)
#define STRINGIFY_ARG   (1 << 2)
#define PASTE_LEFT      (1 << 3)
#define NAMED_OP        (1 << 4)
#define NO_EXPAND       (1 << 5)
#define BOL             (1 << 6)
```

이 구성 요소에 각각에 대해 설명을 달면 아래와 같다.

- PREV_WHITE
  만약 이 token 앞이 whitespace 라면.
- DIGRAPH
  만약 digraph 이었다면.
- STRINGIFY_ARG
  만약 문자열화되어야 하는 macro argument 라면.
- PASTE_LEFT
  만약 ## operator 의 LHS 상이라면.
- NAMED_OP
  C++ named operator 들.
- NO_EXPAND
  이 token 을 macro-expand 하지 않음.
- BOL
  행(줄) 시작에서의 token.

- val
  - cpp_hashnode *node;
    식별자 : identifier
  - const cpp_token *source;
    이 token 으로 부터의 inherit padding.
  - struct cpp_string str;
    문자열 혹은 숫자.
  - unsigned int arg_no;
    CPP_MACRO_ARG 를 위한 argument 갯수.
  - unsigned char c;
    CPP_OTHER 에서 문제시 되는 문자.

이 구조체는 union 을 하나 포함하고 있는데, 이 union 에는 여러 다른 구조체들을 포함하고 있습니다. 각각의 대해서는 struct cpp_string 를 제외하고 모두 위에서 언급하였으니 그것을 참조하시기 바랍니다.

계속해서 cpp_token 구조체의 하위 구조체중 설명하지 않은
struct cpp_string 구조체에 대해서 계속 하겠습니다. 이 구조체는
$prefix/gcc/cpplib.h 파일에 선언되어 있으며, 원형은 아래와 같습니다.

```
struct cpp_string
{
  unsigned int len;
  const unsigned char *text;
};
```

이것은 NUMBER 혹은 STRING, CHAR, COMMENT token 들을 적재 (payload) 하는데 사용됩니다.

## 2.10 tokenrun

tokenrun 구조체에 대해서 살펴보도록 하겠습니다. 이 구조체는
$prefix/gcc/cpphash.h 에 선언되어 있습니다.

```
typedef struct tokenrun tokenrun;
struct tokenrun
{
  tokenrun *next, *prev;
  cpp_token *base, *limit;
};
```

token 들의 수행에 관한 연결 리스트를 가지고 있습니다.

## 2.11 struct deps

struct deps 구조체는 $prefix/gcc/mkdeps.c 파일의 앞부분에 정의되어 있습니다. 함수의 모습은 아래와 같습니다.

```
struct deps
{
  const char **targetv;
  unsigned int ntargets;
  unsigned int targets_size;

  const char **depv;
  unsigned int ndeps;
  unsigned int deps_size;
};
```

이 구조체에 대한 GCC 에서의 설명을 보면 아래와 같이 나와 있습니다.

> "Keep this structure local to this file, so clients don't find it easy to start making assumptions."

제가 실력이 부족하여, 혹은 제가 아직 이에 대한 쓰임새를 격어보지 못하여서 정확하게 설명드릴 수 없음에 죄송스럽습니다. 저도 GCC 세상속의 한 사람일 뿐입니다. Master 가 되는 그날까지.. ㅎㅎ
각 구성요소에 대한 간단한 설명을 하도록 하겠습니다.

- `targetv`
  아직 정확한 설명이 없습니다. 추후 UPDATE 하겠습니다.
- `ntargets`
  실제로 들어차 있는 slot 들의 갯수.
- `targets_size`
  할당된 공간의 양. word 로.
- `depv`
  아직 정확한 설명이 없습니다. 추후 UPDATE 하겠습니다.
- `ndeps`
  아직 정확한 설명이 없습니다. 추후 UPDATE 하겠습니다.
- `deps_size`
  아직 정확한 설명이 없습니다. 추후 UPDATE 하겠습니다.

내부적으로 포함되어 있는 구조체가 없으므로 다음으로 넘어 가겠습니다.

## 2.12 struct pragma_entry

이 구조체는 등록된 pragma 혹은 pragma namespace 를 포함하고 있습니다. 원형은 아래와 같습니다.

```
typedef void (*pragma_cb) PARAMS ((cpp_reader *));
struct pragma_entry
{
  struct pragma_entry *next;
  const cpp_hashnode *pragma;
  int is_nspace;
  union {
    pragma_cb handler;
    struct pragma_entry *space;
  } u;
};
```

이 구조체 또한 이것 저것 다른 하위 구조체들을 포함하고 있지만, 자세히 보면 우리가 위에서 다루었던 것들로 구성되어 있다는 사실을 알 수 있다. Pass!!

## 2.13 struct cpp_callbacks

이제 우리는 전처리기에서 사용하는 Call Back 들을 볼 것입니다. 이 구조체는 $prefix/gcc/cpplib.h 에 선언되어 있습니다. 원형은 아래와 같습니다.

```
struct cpp_callbacks
{
  void (*line_change) PARAMS ((cpp_reader *, const cpp_token *, int));
  void (*file_change) PARAMS ((cpp_reader *, const struct line_map *));
  void (*include) PARAMS ((cpp_reader *, unsigned int,
                           const unsigned char *, const cpp_token *));
  void (*define) PARAMS ((cpp_reader *, unsigned int, cpp_hashnode *));
  void (*undef) PARAMS ((cpp_reader *, unsigned int, cpp_hashnode *));
  void (*ident) PARAMS ((cpp_reader *, unsigned int,const cpp_string *));
  void (*def_pragma) PARAMS ((cpp_reader *, unsigned int));
};
```

각 구성요소에 대해 설명하겠습니다.

- `line_change`
  전처리된 output 의 새로운 줄이 시작될때 마다 호출됩니다.
- `file_change`
  아직 정확한 설명이 없습니다. 추후 UPDATE 하겠습니다.
- `include`
  아직 정확한 설명이 없습니다. 추후 UPDATE 하겠습니다.
- `define`
  아직 정확한 설명이 없습니다. 추후 UPDATE 하겠습니다.
- `undef`
  아직 정확한 설명이 없습니다. 추후 UPDATE 하겠습니다.
- `ident`
  아직 정확한 설명이 없습니다. 추후 UPDATE 하겠습니다.
- `def_pragma`
  아직 정확한 설명이 없습니다. 추후 UPDATE 하겠습니다.

## 2.14 struct ht

이 구조체는 $prefix/gcc/hashtable.h 에 선언되어 있습니다. 이것은 Cpplib 와 Front end 들을 위한 식별자 hash table 을 나타내는데, 원형은 아래와 같습니다.

```
struct ht
{
  struct obstack stack;

  hashnode *entries;
  hashnode (*alloc_node) PARAMS ((hash_table *));

  unsigned int nslots;
  unsigned int nelements;

  struct cpp_reader *pfile;

  unsigned int searches;
  unsigned int collisions;
};
```

각 구성요소의 쓰임새에 대해서 알아봅시다.

- `stack`
  식별자들은 여기서 할당됩니다.
- `entries`
  아직 정확한 설명이 없습니다. 추후 UPDATE 하겠습니다.
- `alloc_node`
  이 구조체와 관련한 Call Back 함수입니다.
- `nslots`
  Entry 배열내의 총 slot 수.
- `nelements`
  (살아있는—활동하는) element 들의 수.
- `pfile`
  만약을 위한 reader 에 대한 링크. cpplib 의 효율적 이득을 위해 사용될 수 있습니다.
- `searches, collisions`
  테이블 사용 통계치에 대해 조사를 할 때 사용됩니다.

이 구조체에는 아직 소개하지 않은 구조체가 하나 존재합니다.

## 2.15 struct hashnode

위에서 포함하고 있는 hashnode 구조체에 대해서 알아보겠습니다. 이 구조체는 $prefix/gcc/tradcpp.c 파일에 선언되어 있습니다. Hash table 내에서의 node 의 구조를 나타냅니다. Hash table 은 #define 명령어 (type T_MACRO) 로 정의된 모든 token 들을 위한 entry 들을 가지고 있습니다. 더군다나 __LINE__ 과 같은 몇몇 특별한 token 들도 가지고 있습니다. (그러한 것들은 그들 자신만의 type 을 가지고 있고 그러한 node 의 type 이 발견 될 때 적당한 code 가 수행됩니다.). 하지만 코드의 여러 부분에서 인식되어진 "#define" 같은 제어 단어들을 포함하지는 않습니다. 이 구조체에 대한 원형은 아래와 같습니다.

```
struct hashnode {
  struct hashnode *next;
  struct hashnode *prev;
  struct hashnode **bucket_hdr;
  enum node_type type;
  int length;
  U_CHAR *name;
  union hashval value;
};
```

각 구성요소의 쓰임새에 대해서 알아 보도록 하겠습니다.

- `next, prev`
  쉬운 삭제를 위해 이중 연결 리스트
- `bucket_hdr`
  또한 node 의 hash chain 을 가르키는 back 포인터를 유지합니다. 그럴 경우 이 node 는 chain 의 head 이며 삭제된 것을 얻습니다. (필자: 설명이 정확하지 않을 수 있습니다.)
- `type`
  secial token 의 type
- `length`
  빠른 비교를 위해 token 의 길이
- `name`
  실제 이름
- `value`
  expansion 혹은 다른 것에 대한 포인터

이 구조체에서 살펴봐야 할 구조체 하나와 열거형 하나가 존재합니다.

### 2.15.1 enum node_type

먼저 열거형부터 알아 봅시다. 설명에서 알 수 있듯이 "secial token 의 type" 을 담고 있다고 합니다. struct hashnode 구조체 부근에서 발견되는 이 열거형은 Hash node 들의 각기 다른 취향들에 대한 정보를 가지고 있습니다. 또한 이것은 keyword table 에서도 사용 가능하다고 합니다. 이 것이 어떻게 선언되어 있는지 한번 모습을 살펴보도록 하겠습니다.

```
enum node_type {
 T_DEFINE = 1,  /* '#define' */
 T_INCLUDE,     /* '#include' */
 T_INCLUDE_NEXT,/* '#include_next' */
 T_IFDEF,       /* '#ifdef' */
 T_IFNDEF,      /* '#ifndef' */
 T_IF,          /* '#if' */
 T_ELSE,        /* '#else' */
 T_ELIF,        /* '#elif' */
 T_UNDEF,       /* '#undef' */
 T_LINE,        /* '#line' */
 T_ENDIF,       /* '#endif' */
 T_ERROR,       /* '#error' */
 T_WARNING,     /* '#warning' */
 T_ASSERT,      /* '#assert' */
 T_UNASSERT,    /* '#unassert' */
```

```
typedef struct definition DEFINITION;
struct definition {
  int nargs;
  int length;
  U_CHAR *expansion;
  struct reflist {
    struct reflist *next;
    char stringify;
    char raw_before;
    char raw_after;
    int nchars;
    int argno;
  } *pattern;
  const U_CHAR *argnames;
};
```

그림 5: struct definition

```
 T_SPECLINE,    /* 특별한 심볼 '__LINE__' */
 T_DATE,        /* '__DATE__' */
 T_FILE,        /* '__FILE__' */
 T_BASE_FILE,   /* '__BASE_FILE__' */
 T_INCLUDE_LEVEL, /* '__INCLUDE_LEVEL__' */
 T_VERSION,     /* '__VERSION__' */
 T_TIME,        /* '__TIME__' */
 T_CONST,       /* '__STDC__' 로 사용된 constant 값.  */
 T_MACRO,       /* '#define' 에 의해 정의된 macro  */
 T_SPEC_DEFINED, /* #if 문장에서 사용하는
                    특별한 'defined' macro.  */
 T_UNUSED       /* 정의되지 않은 것들을 위해 사용됨.  */
};
```

C 언어를 한번 쯤 짜보신 분들은 위의 내용을 보시면 왠지 친숙한 느낌이 드실겁니다. 물론 이 문서를 읽는 분들은 대학교 4 학년 과정을 이수한 경우가 많을 것이지만 말입니다.

### 2.15.2 union hashval

이제 다른 설명되지 않은 구조체 union hashval 에 대해서 알아 보도록 하겠습니다.

이 구조체는 $prefix/gcc/tradcpp.c 파일에 선언되어 있습니다. 이 공동체는 Hash node 의 value field 에서 볼 수 있는 것들의 종류를 가지고 있습니다. 하지만 실제 이것은 이제 거의 쓸모없게 되었다고 합니다. 하지만 이에 대한 추적은 계속됩니다. 우선 원형을 보도록 하겠습니다.

```
union hashval {
  const char *cpval;
  DEFINITION *defn;
  struct answer *answers;
};
```

이에 대한 설명은 아직 많이 부족한 상태입니다. 추후 UPDATE 할 것을 약속 드리며, 여기에 선언된 DEFINITION 구조체에 대해서 알아 보도록 하겠습니다.

이 구조체는 $prefix/gcc/tradcpp.c 파일에 선언되어 있습니다. 이것은 모든 #define 구문을 위해 할당된 구조체입니다. 원형은 그림 5 를 참조하시면 됩니다.

각 구성요소에 대한 쓰임새를 알아 보도록 하겠습니다.

- `nargs`
  인자들의 수.
- `length`
  Expansion 문자열의 길이
- `expansion`
  실제로 대체 문장을 저장하는 장소
- `struct reflist pattern`
  - `next`
    연결 리스트
  - `stringify`
    0 이 아닐 경우 이 arg 는 # operator 에 의해 처리되었음을 을미.
  - `raw_before`
    0 이 아닐 경우 ## operator 가 arg 앞에 있을 경우.
  - `raw_after`
    0 이 아닐 경우 ## operator 가 arg 뒤에 있을 경우.
  - `nchars`
    이 arg 가 발생하기 전에 복사해야 하는 literal char 들의 수.
  - `argno`
    (origin-0) 을 대체할 arg 의 수.
- `argnames`

이 구조체는 모든 #define 구문을 위해 할당된 구조체이며

#define foo bar ,

와 같은 간단한 대체를 위해 사용될 경우 nargs = -1, 이고 'pattern' 목록은 null 이 됩니다. 그리고 expansion 은 대체 문장을 가지게 됩니다. Nargs = 0 은 args 가 없는 함수 모양의 macro 를 의미하며 예를 들면 아래와 같습니다.

#define getchar() getc (stdin) .

인자들이 존재할 경우, expansion 은 한번 처리된 args 들을 가지는 대체 문장을 가지고 있습니다. 그리고 reflist 은 input 으로 부터 어떻게 output 을 만들지를 표현하는 목록을 가지고 있습니다. 예를 들면, "3 번째 문자는 첫번째 인자, 9 번째 문자는 세번째 인자, 0 번째 문자는 두번째 인자". 이 문자들은 expansion 으로 얻어온 정보입니다. 마지막 arg-발생 후에 expansion 이 남아 있는 것에 상관없이 그 인자 뒤에 복사가 됩니다. reflist 은 임의의 크기를 가질 수 있다는 사실을 알아 두십시오. 그것의 length 는 대체 문장에서 발견한 인자들이 몇 번 보이는지에 상관합니다. 얼마나 많은 인자들이 존재하는지 하고는 상관 없습니다. 예를 들면 :

#define f(x) x+x+x+x+x+x+x

는 대체 문장으로 "++++++" 를 가질 것이고 pattern 목록으로

(0, 1), (1, 1), (1, 1), ..., (1, 1), NULL

를 가질 것입니다. 여기서 (x, y) 는 (nchars, argno) 를 의미합니다.

## 제 3 절 6 주째 강의를 마치며

[1] 지금 이라크에서는 무구한 사람들이 전쟁으로 인해 이 시간에도 죽어가고 있습니다. 이 전쟁의 목적이 논리가 맞지 않으며 유엔 안전보장 이사회에서 조차도 인증을 받지 않은 미국과 영국의 독단적인 행동으로 이 세계의 평화가 심각하게 위협받고 있는 듯합니다. 제가 태어난 이 나라는 아직까지는 평화로와 이렇게 GCC 에 대한 글도 쓸 수 있지만 우리나라가 아닌 다른 나라의 사람들이 글을 쓰고 있는 이 시간에도 물, 식량, 의료품 부족으로 혹은 미국이 그토록 자랑하는 대량 살상 무기로 부터 죄없는 사람들이 죽어가고 있다고 생각하니 여간 찝찝(-_-; 표현이 좀...)한게 아닙니다. 제가 할 수 있는 일 혹은 우리가 할 수 있는 일이 어떤 것이 있을 지 적극적으로 생각해야 할 단계인 것 같습니다. 사람은 능동적이여야 합니다. [2] 구조체가 상당히 큰데다가 포함해야 하는 다른 내용들도 많아져서 다른 때 보다 시간도 많이 걸리고, 페이지 수도 몇 장 더 많군요. 그림도 한장 만들어서 넣고 싶은데, 조금 아쉬움이 남습니다. 힘들게 6 주차 강의를 마칩니다. 행복하세요.