기반 작업
(2) struct cpp_reader 의 초기화

정원교

2004년 2월 23일

목 차

제 1 절 7 주차 강의를 시작하며

싱그러운 봄 햇살이 비추는 이날 여러분들은 봄 나들이라도 다녀 오셨나요? 저는 그런 사소한 것도 챙기지 못해 마누라에게 바가지 긁히고 있습니다. 여자친구나 와이프를 가지고 계신 분들은 꼭 봄에는 봄 나들이 다녀 오시길 추천합니다.

이번 주의 주제는 제목에서도 알 수 있듯이 “struct cpp_reader 의 초기화” 입니다. 앞 주에서 살펴본 struct cpp_reader 구조체에 대한 GCC 내부에서의 수행들을 살펴보도록 하겠습니다. 이번 주는 그렇게 양이 많지 않을 것 같습니다.

제 2 절 초기화

앞 주에서 이미 말씀드렸듯이 struct cpp_reader 구조체는 GCC 가 실행된 후 *lang_hooks.init_options () 의 수행에 의해 초기화가 됩니다. 내부적으로 C 언어일 경우 c_common_init_options 함수를 호출하게 되며 이 함수 내에서
cpp_create_reader 함수를 호출함으로써 초기화가 이루어 지게 됩니다.

2.1 1 단계: cpp_reader 공간을 할당

sizeof (cpp_reader) 만큼의 공간을 할당합니다. 당연히 xmalloc 과 같은 메모리 할당 함수를 사용할 것입니다.

2.2 2 단계: 언어에 따른 내부 flag 설정

이 부분에 대한 설정은 $prefix/gcc/cppinit.c 파일에 선언되어 있는 set_lang 함수에 의해서 수행하게 됩니다.

이 내부를 보게 되면 struct lang_flags 구조체를 사용하고 있는데, 이것은 주어진 언어에 맞게 내부 flag 들을 설정하는데 사용됩니다. 이에 대한 원형은 아래와 같습니다.

```
static const struct lang_flags lang_defaults[] =
{ /*              c99 objc c++ xnum trig dollar c++comm digr  */
  /* GNUC89 */  { 0,  0,   0,  1,   0,   1,     1,      1     },
  /* GNUC99 */  { 1,  0,   0,  1,   0,   1,     1,      1     },
  /* STDC89 */  { 0,  0,   0,  0,   1,   0,     0,      0     },
  /* STDC94 */  { 0,  0,   0,  0,   1,   0,     0,      1     },
  /* STDC99 */  { 1,  0,   0,  1,   1,   0,     1,      1     },
  /* GNUCXX */  { 0,  0,   1,  1,   0,   1,     1,      1     },
  /* CXX98  */  { 0,  0,   1,  1,   1,   0,     1,      1     },
  /* OBJC   */  { 0,  1,   0,  1,   0,   1,     1,      1     },
  /* OBJCXX */  { 0,  1,   1,  1,   0,   1,     1,      1     },
  /* ASM    */  { 0,  0,   0,  1,   0,   0,     1,      0     }
};
```

각각의 언어마다 cpp_reader 구조체 내부에 설정되는 값들이 조금씩 달라지게 된다는 것을 아실 수 있을 것입니다. 하지만 우리가 살펴보고 있는 언어는 C 이기 때문에, 당연히 위 부분에서 GNUC89 관련 부분이 구조체에 설정될 것입니다.

cpp_reader 구조체에 위의 값들이 아래와 같이 설정이 되게 됩니다. 아래에서 부터 계속 보게될 pfile 변수는 struct cpp_reader 를 가르키는 포인터입니다.

```
CPP_OPTION (pfile, lang) = lang;

CPP_OPTION (pfile, c99)                = l->c99;
CPP_OPTION (pfile, objc)               = l->objc;
CPP_OPTION (pfile, cplusplus)          = l->cplusplus;
CPP_OPTION (pfile, extended_numbers)   = l->extended_numbers;
CPP_OPTION (pfile, trigraphs)          = l->trigraphs;
CPP_OPTION (pfile, dollars_in_ident)   = l->dollars_in_ident;
CPP_OPTION (pfile, cplusplus_comments) = l->cplusplus_comments;
CPP_OPTION (pfile, digraphs)           = l->digraphs;
```

우선 이 부분을 설정하기에 앞서 CPP_OPTION 매크로에 대해서 잠시 언급을 하면 아래와 같은 모습을 가진 것입니다.

```
#define CPP_OPTION(PFILE, OPTION) ((PFILE)->opts.OPTION)
```

이 것은 cpp_reader 구조체 내부에 있는 struct cpp_options 를 설정하게 된다는 것을 알실 것입니다. 그 내부의 각각의 옵션들에 대한 설명은 앞 강의에서 하였기 때문에 언급하지 않겠습니다.

대부분의 GCC 는 프로그램이 복잡해져 감에 따라 포인터에 따른 혼동을 줄이기 위해서 포인터를 통한 직접 접근을 거의 금하고 있습니다. 대신 위와 같이 매크로를 사용함으로써 코드의 가독성과 포인터 남용에 따른 폐단을 줄이고자 시도하고 있습니다.

2.3 3 단계: 그 외의 옵션들 설정

이 부분에서는 아직 설정되지 않은 나머지 옵션 (struct cpp_options) 들에 대하여 설정을 하게 됩니다. 이 부분에서 설정되는 아래와 같습니다.

```
CPP_OPTION (pfile, warn_import) = 1;
CPP_OPTION (pfile, discard_comments) = 1;
CPP_OPTION (pfile, show_column) = 1;
CPP_OPTION (pfile, tabstop) = 8;
CPP_OPTION (pfile, operator_names) = 1;
CPP_OPTION (pfile, signed_char) = 1;
```

각 옵션에 대한 자세한 설명은 이 전 강의록에서 찾으시기 바랍니다. 이 부분에서 봐야 할 부분은 signed_char 옵션을 어떻게 설정할 지를 고민을 해야 합니다. 대부분의 컴파일러가 char 를 signed 형태로 인식하지만 어떤 컴파일러 혹은 환경에서는 unsigned 로 인식할 수 있습니다.

기본적인 값으로 char 는 signed 로 선언됩니다.

2.4 4 단계: struct deps 의 초기화

struct cpp_reader 구조체의 deps 요소를 초기화합니다. 이 부분은 deps_init () 함수내에서 수행됩니다. 실제로 컴파일을 수행하다 보면 이 요소를 사용할 경우도 있고 사용하지 않을 때도 있지만 이 단계에서 이 부분을 생성해 놓는 것이 이 부분을 처리하는 가장 간단한 방법입니다.

deps_init () 함수는 $prefix/gcc/mkdeps.c 파일에 선언되어 있습니다. 이 부분을 실제 보시면 아시겠지만 아주 간단한 수행만 하고 끝나는 기본적인 초기화 부분입니다.

1. struct deps 구조체를 할당합니다.
2. 각 요소를 0 혹은 NULL 로 초기화합니다.

2.5 5 단계: struct line_maps 의 초기화

이 부분 또한 위 4 단계처럼 큰 차이가 없습니다. struct line_maps 의 각 요소들을 0 혹은 NULL 로 초기화합니다. 이 초기화가 이루어지는 부분은 init_line_maps () 함수에서 수행되어 지며 이는 $prefix/gcc/line-map.c 파일에 선언되어 있습니다.

하지만 last_listed 와 trace_includes 요소는 각각 -1, false 로 설정합니다.

그리고 struct cpp_reader 구조체의 line 요소를 1 로 설정합니다. 논리적 줄번호 1 에서 시작하기 때문에 우리는 special 상태들을 위한 줄번호 0 을 사용할 수 있습니다.

2.6 6 단계: lexer state 를 초기화

이 단계에서는 struct cpp_reader 구조체의 state 요소 중 save_comments 를 설정하게 되는데, 앞 강의에서 설명이 되어 있지만 이 요소는

```
CPP_OPTION (pfile, discard_comments);
```

에 영향을 받게 되어 이 값과 반대 값을 항상 가지게 됩니다.

2.7 7 단계: static token 들을 설정

현재 이 구조체에서 고정적인 token 들로는 date, time, avoid_paste, eof 등이 존재하는데, 이 중 time 을 제외한 3 개의 변수를 아래와 같이 설정하게 됩니다. 설정되는 값은 아래와 같습니다.

```
pfile->date.type = CPP_EOF;
pfile->avoid_paste.type = CPP_PADDING;
pfile->avoid_paste.val.source = NULL;
pfile->eof.type = CPP_EOF;
pfile->eof.flags = 0;
```

여기서 cpp_token 인 date, avoid_paste, eof 의 요소인 type 과 val.source, flags 들이 설정되게 되는데 이 부분의 역활에 대해서는 앞 부분에 업근하였습니다. 문제는 val.source 인데, 여기서 val 는 공용체 (union) 으로 선언되어 있기 때문에 각각의 type에 따라서 공용체의 선택이 달라지게 됩니다. 이 부분에 대해서는 따로 다른 시간을 활용해서 설명을 해야 할 듯합니다.

2.8 8 단계: lexer 를 위한 token buffer 생성

이제 lexer 를 위한 buffer 를 생성해야 할 단계입니다. 이 단계에서 버퍼 할당 역활은 $prefix/gcc/cpplex.c 파일에 선언되어 있는 _cpp_init_tokenrun () 함수에서 처리하게 되며 이 부분에서 메모리가 정해진 count 만큼의 공간을 할당하게 됩니다. 그리고 tokenrun 구조체의 구성요소인 base, limit, next 가 설정되게 됩니다. 이 부분은 나중에 그림으로 간단히 표현하도록 하겠습니다.

2.9 9 단계: base context 초기화

여기에서는 큰 함수 수행은 존재하지 않으며 단순히 아래와 같은 수행을 합니다.

1. struct cpp_reader 구조체의 base_context 의 주소를 같은 struct cpp_reader 구조체의 구성요소인 context 에 삽입합니다.
2. base_context 구조체의 구성 요소인 macro 를 0 으로 설정합니다.
3. base_context 구조체의 구성 요소인 prev, next 를 0 으로 설정합니다.

2.10 10 단계: 정렬, 비정렬 저장소 초기화

정렬 저장소와, 비정렬 저장소를 초기화합니다. 이 부분은 $prefix/gcc/cpplex.c 파일에 선언되어 있는 _cpp_get_buff () 함수에서 이루어 집니다.

struct cpp_reader 구조체의 요소인 a_buff, u_buff, free_buffs 중에서 free_buffs 가 이러한 buffer 관리를 하게 됩니다. 내부적으로 free_buffs 의 내용을 순환하면서 충분히 큰 buffer 를 반환하지만 쓸모없이 낭비하지 않습니다. 만약 할당된 free_buffs 가 아직 존재하지 않는다면 new_buff () 함수에서 새로운 buffer 를 할당하게 됩니다.

new_buff () 함수는 당연히 메모리 공간을 할당하는 함수이지만, 조금 살펴봐야 할 부분이 있씁니다. 내부에서 사용하는 매크로에 대한 것입니다.

다음의 새 상수의 변경으로 performance 에서 놀라운 효과를 가져올 수 있습니다. 여기 값들은 나름대의 이유로 이렇게 설정되었지만 좀 더 조절 (tune) 해야 할 필요가 있습니다. 만약 여러분이 이것을 조절하게 된다면 cpplib 사용의 전반적인 부분에서 검사를 해야하며 heavy nested function-like macro expansion 도 포함하여 검사하셔야 합니다. 또한 메모리 사용에 관한 변화 또한 검사하여야 합니다. (NJAMD 가 이것을 검사하는데 좋은 도구입니다.

```
#define MIN_BUFF_SIZE 8000
#define BUFF_SIZE_UPPER_BOUND(MIN_SIZE) \
        (MIN_BUFF_SIZE + (MIN_SIZE) * 3 / 2)
#define EXTENDED_BUFF_SIZE(BUFF, MIN_EXTRA) \
        (MIN_EXTRA + ((BUFF)->limit - (BUFF)->cur) * 2)
```

위의 값에서 MIN_BUFF_SIZE 는 BUFF_SIZE_UPPER_BOUND (0) 보다 절대 크면 안됩니다. 살펴봐야 할 또 다른 매크로가 몇 개 있습니다.

```
struct dummy
{
  char c;
  union
  {
    double d;
```

```
    int *p;
  } u;
};

#define DEFAULT_ALIGNMENT (offsetof (struct dummy, u))
#define CPP_ALIGN(size, align) \
        (((size) + ((align) - 1)) & ~((align) - 1))
```

위의 매크로를 사용하여 인자 len 을 alignment 시키게 됩니다. 4 의 배수값을 가지게 됩니다.

2.11 11 단계: buffer obstack 초기화

이 부분은 gcc_obstack_init () 함수를 사용하여 obstack 를 초기화하게 됩니다. 이 함수는 $prefix/libiberty/hashtable.c 파일에 선언되어 있습니다. 하지만 libiberty 라이브러리 쪽으므로 여기에서는 다루지 않겠습니다. 이 부분에 대한 설명서는

http://gcc.gnu.org/onlinedocs/

에서 찾으실 수 있으니, 그 쪽을 참조하시기 바랍니다. 저 또한 조만간 libiberty 에 관한 강의글을 올리도록 하겠습니다.

2.12 12 단계: all_include_files 초기화

이 컴파일을 통해서 보게 될 모든 파일 이름에 관한 정보를 저장하는데 사용되는 splay tree 를 설정합니다. 또한 open 을 시도하였지만 실패 한 각 파일에 관한 entry 도 가지고 있습니다. – 이것은 나중에 compiler 가 그것을 열려고 시도하지 않을 것이기 때문에 system call 을 절약할 수 있습니다.

각 node 의 key 는 _cpp_simplify_pathname 에 의해 처리된 후의 파일이름입니다. 경로 이름은 절대적일 수 있고 아닐 수 있습니다. 경로 문자열은 malloc 를 통해서 할당되며 splay tree key deletion function 으로 지정되는 free () 함수에 의해서 자동으로 free 될 것입니다.

node 의 값은 struct include_file 를 가르키는 포인터이며 절대 NULL 이 아닙니다.

이 부분에서 사용하는 splay_tree_new () 함수 또한 libiberty 라이브러리의 한 부분입니다. 이것도 나중 강의에서 설명하도록 하겠습니다.

제 3 절 7 주째 강의를 마치며

이번 강의에는 꼭 그림 한장을 넣어봐야지 라고 생각을 하고서 여러가지 방법을 생각하고 해보고 했는데, 결국에는 실패군요. PSTricks 나 MetaPost 등의 사용을 시도해 보았지만, 결코 만만치 않은 인터페이스와 그림 한장 그리는데, 소유되는 시간 또한 상당한 것 같아 이번에도 넣지 못했습니다. 사용방법에 있어서 가장 손쉬운 것은 역시 Graphviz 가 아닐까 쉽군요. 하지만 이것도 저의 2%를 채워주지 못하는 군요.

이렇게 7 주째 강의는 빨리 씁니다.