**과목명: 시스템프로그래밍**

**2 분반**

**<<Project #2>>**

**서강대학교 [컴퓨터공학과]**

**[20161566]**

**[권형준]**

목 차

1. **프로그램 개요**
2. **프로그램 설명**
   1. 프로그램 흐름도
   2. 흐름도 설명
3. **(새로 추가 된) 모듈 정의**
   1. Void Type()
   2. Void Assemble()
   3. Void Assemble\_error\_phase\_1(FILE\* fp, FILE\* fp2, char a[], int line)
   4. Void Assemble\_error\_phase\_2(FILE\* fp, FILE\* fp2, FILE\* fp3, char a[], char b[], char c[], int line)
   5. int Get\_Location(char target[], char save[], int location, int type);
   6. int Add\_Symbol(char target[], int location);
   7. int Check\_Symbol\_Location(char target[]);
   8. int Get\_Format(char target[])
   9. int Get\_Register(char target[])
   10. int Get\_Opcode(char target[])
   11. void Symbol()
4. **(새로 추가된) 전역 변수 정의**
   1. Struct symbol\_node
   2. Symbol\_node\* temp\_symbol\_head
   3. Symbol\_node\* symbol\_head
5. **코드 설명**

* **<20161566.c>**
* **<20161566.h>**
* **Makefile**

1. **프로그램 개요**

이전의 프로젝트에서 구현한 shell에 assemble 기능을 추가하는 프로그램이다. SIC/XE의 assembly program source 파일(.asm)파일을 입력 받아서 object 파일(.obj)과 list 파일 (.lst)파일을 생성한다. 또한 과정에서 생성된 symbol table과 파일의 내용을 확인할 수 있는 type()명령어까지 구현한다.

1. **프로그램 설명**
   1. **프로그램 흐름도(새로 추가된 부분)**

**텍스트, 지도이(가) 표시된 사진

자동 생성된 설명**

* 1. **흐름도 설명**

흐름도를 간단히 설명하자면, 지난 프로젝트 1에서 사용한 흐름도에서 명령어가 3개 추가되었다는 점에서 출발하였다. Check\_Command()함수는 명령어에 알맞은 함수를 호출한다. 위의 흐름도에서 Type(), Assemble(), Type()함수가 새로 추가된 3개의 명령어를 수행한다. Symbol과 type 함수는 실행도중 오류가 발생하면 오류를 출력하고 성공하면 add\_history함수를 호출한다. Assemble함수는 우선 파일을 열고, 성공적으로 열었다면 phase1을 거치게 된다. 이 과정에서 temp파일을 생성하고, phase2에서는 이 temp파일을 사용하여 .obj와 .lst를 만든다. Phase1에서는 symbol의 중복과 instruction의 유효성을 판단한다. 이때 오류가 발생하면 assemble\_error\_phase\_1을 통해 현재 열러 있는 temp파일과 asm파일을 닫고 temp파일을 삭제한 뒤 동적으로 할당된 temp\_symbol\_table을 free한다. Phase2에서는 operand로 들어오는 값들의 유효성을 확인한다. 이때 오류가 발생하면 assemble\_error\_phase\_2를 호출하고, 이 역확은 assemble\_error\_phase\_1와 의미가 같다. Assemble이 성공적으로 완료되면 성공을 출력하고 add\_history함수를 호출한다.

1. **모듈 정의**
   1. Void Type()

변수 정의

🡪Char temp: 출력할 문자

🡪Char last\_char: 마지막 문자를 저장한다.

🡪FILE \*fp: 출력할 파일

함수 설명

🡪파일이 존재하지 않으면 오류를 출력한다. 만약 파일이 존재한다면 character하나씩 읽어와서 출력을 한다. 마지막으로 출력한 character가 ‘\n’, 즉 개행 문자가 아니었다면 개행 문자를 추가로 출력한다. 파일을 닫고 Add\_History를 호출하여 history에 실행을 저장한다.

* 1. Void Assemble()

변수 정의

🡪int line: 줄의 번호를 저장한다.

🡪int LOCCTR: 각 줄의 주소, location을 저장한다.

🡪int base: base의 주소를 저장한다.

🡪int PC: 현재 PC의 위치를 저장한다.

🡪int code\_length: 총 주소의 길이를 저장한다.

🡪int biggest\_size: 각 줄을 입력 받고, 가장 긴 줄의 길이를 저장한다.

//lst파일 생성시 사용되는 변수//

🡪int location\_save[]: LOCCTR의 값들을 순서대로 저장한다.

🡪int location\_save\_size: location\_save의 크기를 저장한다.

🡪int last\_location: PC값을 구하기 위해 사용된다

//obj파일 생성시 사용되는 변수//

🡪int obj\_last\_address: obj파일 생성시 줄의 길이를 조절하기 위한 변수

🡪int modification[]: format4형식에서 modification이 필요한 주소를 저장하는 변수.

🡪int modification\_size: modification의 크기를 저장

//파일 관련 변수//

🡪char\* list\_outputfile: lst파일

🡪char\* obj\_temp\_outputfile: 임시 obj파일

🡪char\* obj\_outputfile: obj파일

🡪char\* temp\_outputfile: 임시 파일

🡪char\* extender: .asm파일을 확인하기 위한 문자열

🡪int size: 파일에서 읽은 각 줄의 길이를 저장한다.

🡪char file\_input[MAX\_INPUT\_SIZE]: 파일에서 읽은 각 줄을 저장한다.

🡪char token[4][MAX\_TOKEN\_SIZE]: 줄을 토큰으로 나눈 것을 저장한다.

🡪int token\_num: 나눈 토큰을 저장한다.

🡪char temp\_token[6][MAX\_TOKEN\_SIZE]: 임시 토큰을 저장한다.

함수 설명

🡪파일이 존재하지 않으면 오류를 출력한다. 만약 존재한다면 우선 lst, obj, temp파일의 이름을 만들어 둔다.

처음 phase1을 시작하기 위해서 temp파일을 쓰기로 열고 asm파일을 읽기로 연다. Phase1은 asm파일을 한 줄씩 읽어서 더 이상 읽을 줄이 없다면 종료하게 된다. 줄을 읽은 후 만약 줄의 끝에 개행 문자뿐만 아니라 ‘Carriage return’값이 있다면 이를 처리하기 위해 size-2를 한다. 이제 읽은 줄을 token으로 나눠서 token[][]에 저장한다. 처음 token을 확인하여 만약 주석이라면 line을 증가시키고 temp파일에 그대로 출력한 뒤 다음 줄을 읽는다. 만약 주석이 아니라면, 처음 줄을 확인하고 START명령어가 제대로 있는지 확인한다. 없다면 오류를 출력한다. Phase1에서 오류를 출력한다는 것은 Assemble\_error\_phsae\_1을 호출한다는 의미이다. 다음 START가 제대로 있다면 다음 줄을 읽는다. 읽은 줄의 token의 개수에 따라서 경우의 수를 나눈다. 토큰의 개수가 0개라면 그대로 출력하고 다음 줄로 넘어간다. 토큰의 개수가 1개 또는 2개라면, 이는 instruction만 있는 경우, instruction과 operand만 있는 경우에 속한다. Get\_Location을 사용하여 instruction과 operand에 의해 증가된 LOCCTR주소를 구하고 temp파일에 출력한다. 마약 토크의 개수가 3개라면 symbol이 추가된 instruction이거나, symbol없이 operand가 2개(예: BUFFER, X)와 같은 것들이다. 두 경우를 나누어 처리하고 symbol의 경우 Add\_Symbol함수를 사용하여 symbol이 중복되어 사용되는지 확인한다. 만약 중복되어 사용되지 않았다면 symbol의 주소와 같이 temp\_symbol\_table에 저장한다. 토큰의 개수가 4개 이상이라면 symbol. Instruction이 있고 인자도 2개인 경우이다. 이러한 경우에도 Add\_symbol로 확인하고 Get\_Location을 사용하여 증가된 LOCCTR을 구하여 출력한다. 이외의 경우는 오류를 출력한다. 이때 성공적으로 증가한 LOCCTR은 location\_save에 저장한다. Location\_save는 나중에 PC를 구하는데 사용된다. temp파일에서 한 줄을 읽은 후 PC는 다음 줄의 LOCCTR을 가리키게 되는데 그때마다 다음 LOCCTR을 구하는 일은 phase1을 의미 없게 만듦으로 LOCCTR을 미리 저장한다.

다음 phase2를 실행하기 앞서 asm파일과 temp파일을 닫고 temp파일을 읽기 형식으로 lst, 임시 obj파일을 쓰기형식으로 연다. 또한 오류 출력을 위해 line을 다시 5로 초기화 한다. Phase1과 동일하게 token으로 나눈다. 하지만 이때는 token[0], token[1], token[2]에 저장되는 값의 종류를 동일하게 한다. Token[0]에는 instruction을 token[1],[2]에는 operand의 값을 저장한다. 이렇게 하는 이유는 더 이상 앞의 symbol이 의미가 없기 때문이다. 다음 각 줄 별로 instruction과 operand를 사용하여 operand가 존재하지 않거나 유효하지 않으면 오류를 출력하고 그 외의 경우에는 instruction의 format또는 의미를 사용하여 object code를 완성한다. Format 3,4에서의 nixbpe는 결국 4bit씩 묶여서 16진수로 표현되기 미리 4bit씩 나눠서 (예를 들어서 xbpe는 하나로 묶인다) 각 자리별로 8, 4, 2, 1의 값을 부여하여 더한다. (xbpe가 1010이라면 xbpe의 값은 10이 되고 16진수로 A가 된다. 이는 object코드 3번째 자리에 출력한다) 그리고 object코드를 lst파일에서는 temp파일 뒤에, 임시 obj파일에서는 형식에 맞게 쭉 출력한다. 이때, MAX\_OBJ\_LINE의 값보다 object코드의 길이가 길어지면 다음 줄에 계속 출력한다. 임시 obj파일에 출력하는 이유는 object코드를 나열하는 구간보다 그 구간의 길이가 먼저 와야 되는데, 그 길이를 아직 알 수 없기 때문이다. 모든 줄이 위의 과정을 거치게 된 후 임시 obj파일에 modification할 주소들을 적고 마지막 End를 출력한다.

마지막 phase는 phase 2의 일부로 임시obj파일을 obj파일로 변형시킨다. 현재 임시 obj파일에서 수정할 부분은 T로 시작하는 부분이다. 이 부분의 앞은 올바른 형태이지만 첫 주소와 연속된 object코드 사이에 길이가 입력 되어있지 않다. 한 줄씩 읽어서 연속된 object코드의 길이를 구해 그 사이에 넣어서 출력해 최종 obj파일을 생성한다.

* 1. Void Assemble\_error\_phase\_1(FILE\* fp, FILE\* fp2, char a[], int line)

변수 정의

🡪symbol\_node\* temp3: temp\_symbol\_table을 free하는데 사용된다.

함수 설명

🡪Assemble phase1에서 오류가 발생하면 현재 열려있는 asm파일과 temp파일을 닫고 temp파일은 삭제하는 함수이다. 또한 그동안 동적으로 할당하여 저장한 symbol들의 값을 free해준다.

* 1. Void Assemble\_error\_phase\_2(FILE\* fp, FILE\* fp2, FILE\* fp3, char a[], char b[], char c[], int line)

변수 정의

🡪

함수 설명

🡪Assemble phase2에서 오류가 발생하면 호출된다. Phase1과 마찬가지로 phase2 도중에 열려 있는 파일들을 닫고 추가로 생성한 lst파일, obj파일, temp파일을 삭제한다. 또한 temp\_symbol\_table을 free한다.

* 1. int Get\_Location(char target[], char save[], int location, int type)

변수 정의

🡪int compare\_location: 기존의 location과 최종 location을 비교하여 변동을 확인.

🡪int flag: format 4인 instruction을 구분한다.

함수 설명

🡪instruction과 뒤의 인자를 받아서 증가한 LOCCTR값을 반환한다. 성공적으로 반환하지 못하는 경우, instruction이 없거나 증가한 LOCCTR이 범위를 벗어나는 경우, -1을 반환한다. BYTE의 경우에는 C, X를 구분하여 증가시킨다. Word 또는 byte 변수가 들어온 경우 뒤의 인자까지 확인하여 location을 증가시킨다. 마지막 mnemonic중 하나라면 hash\_table에서 해당하는 format을 찾아서 그만큼 증가시킨다.

* 1. int Add\_Symbol(char target[], int location)

변수 정의

🡪symbol\_node\* temp: temp\_symbol\_table을 돌기 위한 노드

함수 설명

🡪temp\_symbol\_table을 돌면서 해당하는 symbol이 중복되어 있는지 확인한다. 또한 strcmp를 사용하여 그 값이 0보다 크면 next로 넘어가 애초에 linked list를 만들 때 내림차순으로 정리한다. 만약 중복되는 symbol이 있다면 0을 반환하고 제대로 저장이 되면 1을 반환한다.

* 1. int Check\_Symbol\_Location(char target[])

함수 설명

🡪symbol을 받아서 해당하는 location을 반환한다. 만약 아직 temp\_symbol\_table이 만들어지지 않았거나 해당하는 symbol이 존재하지 않는다면 -1을 return한다.

* 1. Int Get\_Format(char target[])

변수 정의

🡪int hash: target을 hash값으로 변환하여 저장한다.

함수 설명

🡪 mnemonic을 입력 받아 그 format을 찾아 반환 한다. 만약 없다면 -1을 반환한다.

* 1. int Get\_Register(char target[])

함수 설명

🡪target으로 들어온 register값이 만약 SIC/XE머신에서 사용하는 기본적인 9개의 register 중 하나라면 그에 해당하는 register number를 return한다. 존재하지 않는다면 -1을 반환한다.

* 1. int Get\_Opcode(char target[])

함수 설명

🡪mnemonic에 해당하는 format을 반환한다. 존재하지 않는다면 -1을 반환한다.

* 1. void Symbol()

함수 설명

🡪Symbol명령어를 수행한다. Symbol\_table이 없다면, 즉 아직 assemble명령어가 성공적으로 수행된 적이 없다면 오류를 출력한다. 만약 저장된 값이 있다면 이미 linked\_list에 내림차순으로 저장되어 있기 때문에 head부터 출력한다. Add\_History를 호출한다.

1. **전역 변수 정의**
   1. Struct symbol\_node

🡪char symbol: symbol을 저장한다.

🡪int loc: 그 symbol이 가리키는 주소를 저장한다.

🡪struct symbol\_node\* next: linked list를 생성하기 위해 다음 노드를 가리킨다.

Symbol명령어 수행, 그리고 phase2에서 operand에 해당하는 symbol의 주소를 찾기 위해 만드는 linked list의 구성 node이다.

* 1. Symbol\_node\* temp\_symbol\_head

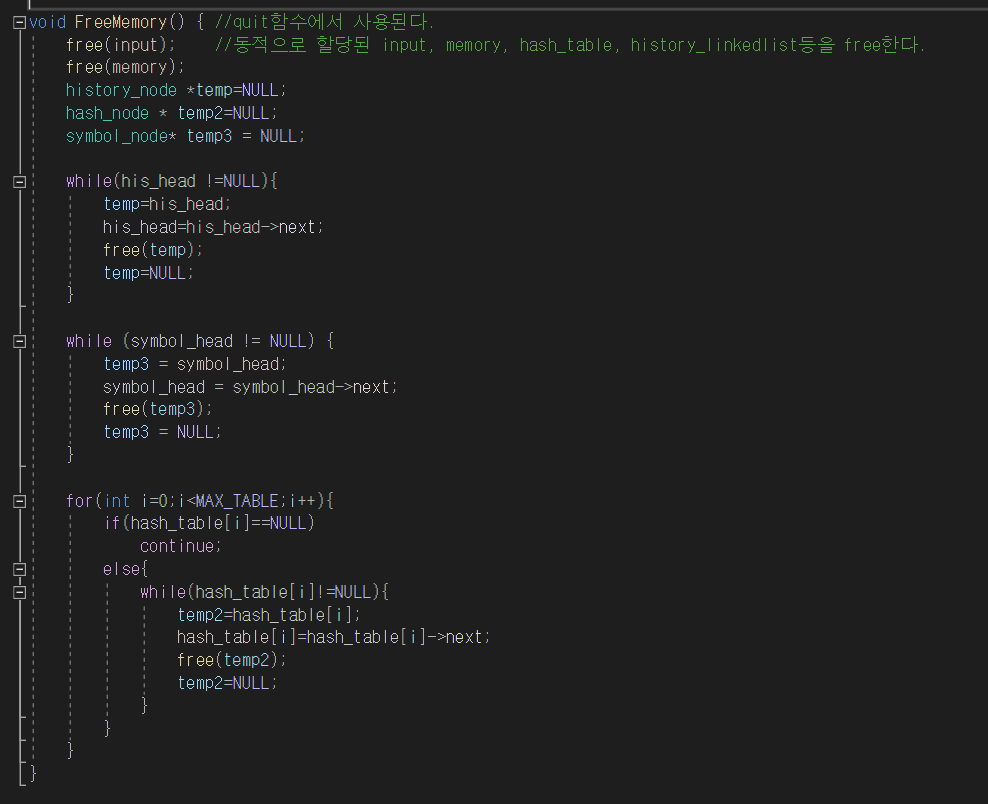
🡪Assemble 명령어 수행 도중 error가 발생하면 도중에 저장된 temp\_symbol\_table은 삭제하고 원래 있던 symbol\_table은 유지해야 되기 때문에 임시 symbol\_table을 만들어야 된다. 그 linked\_list의 머리를 가리키는 변수.

* 1. Symbol\_node\* symbol\_head

🡪assemble이 성공적으로 수행된 이후 저장되는 symbol table의 머리를 가리킨다. quit명령어로 종료하기 전에 freememory에서 free의 과정을 거친다.

1. **코드 설명(새로 추가된 부분)**

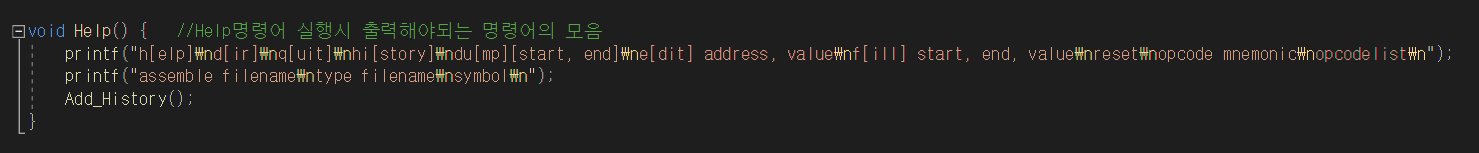
**<20161566.c>**



.

.

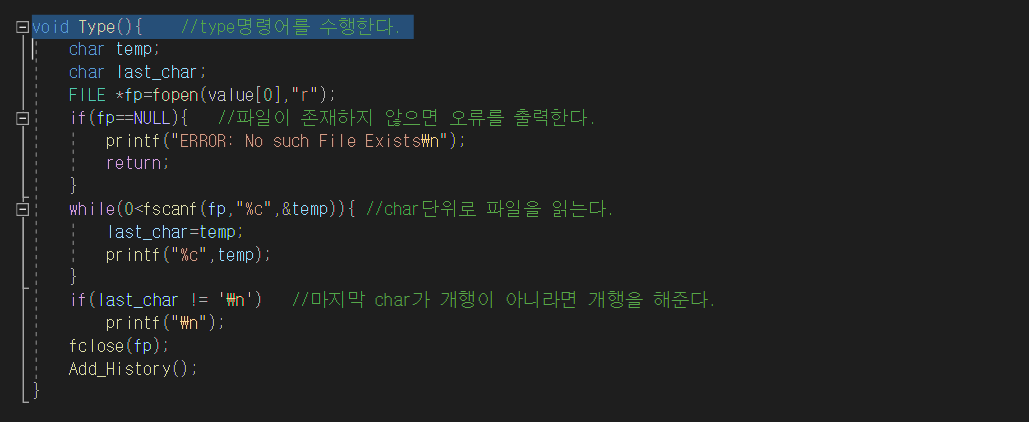
.

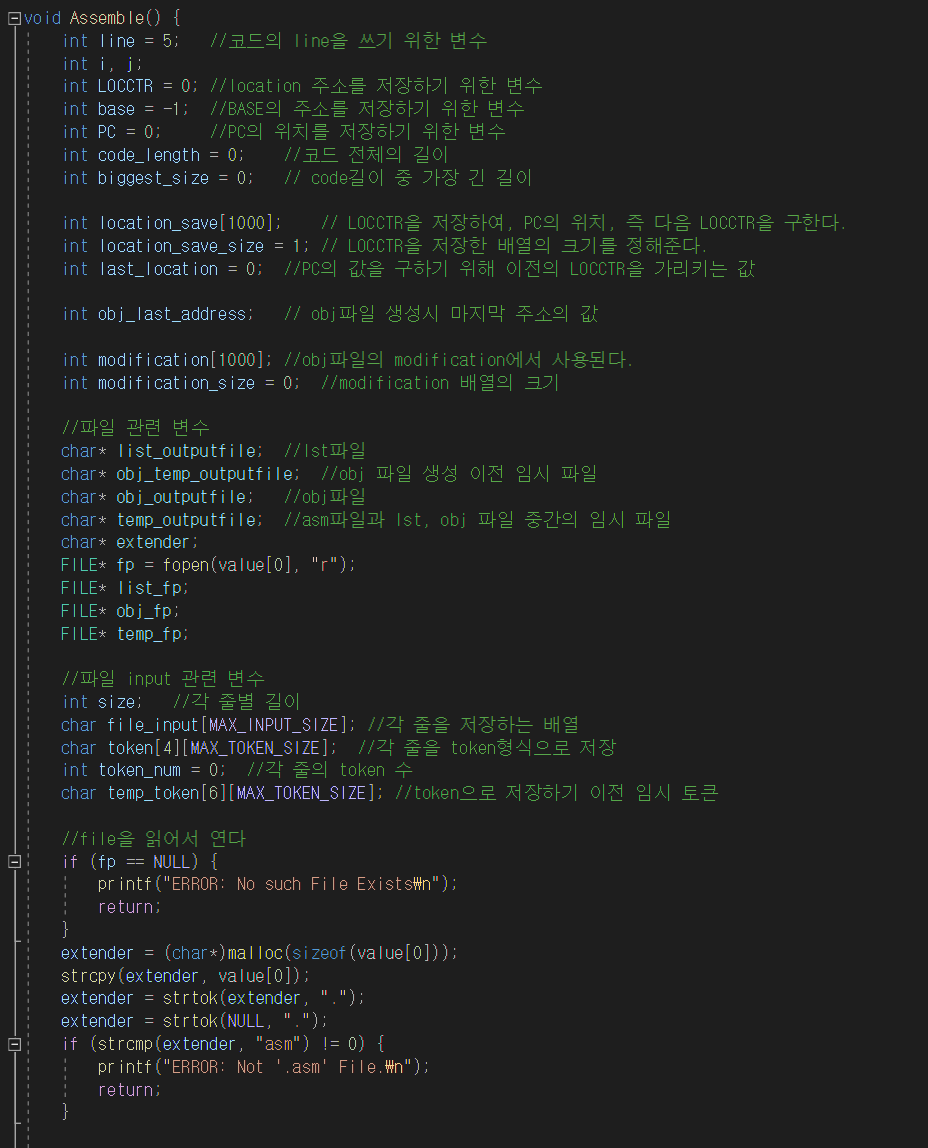


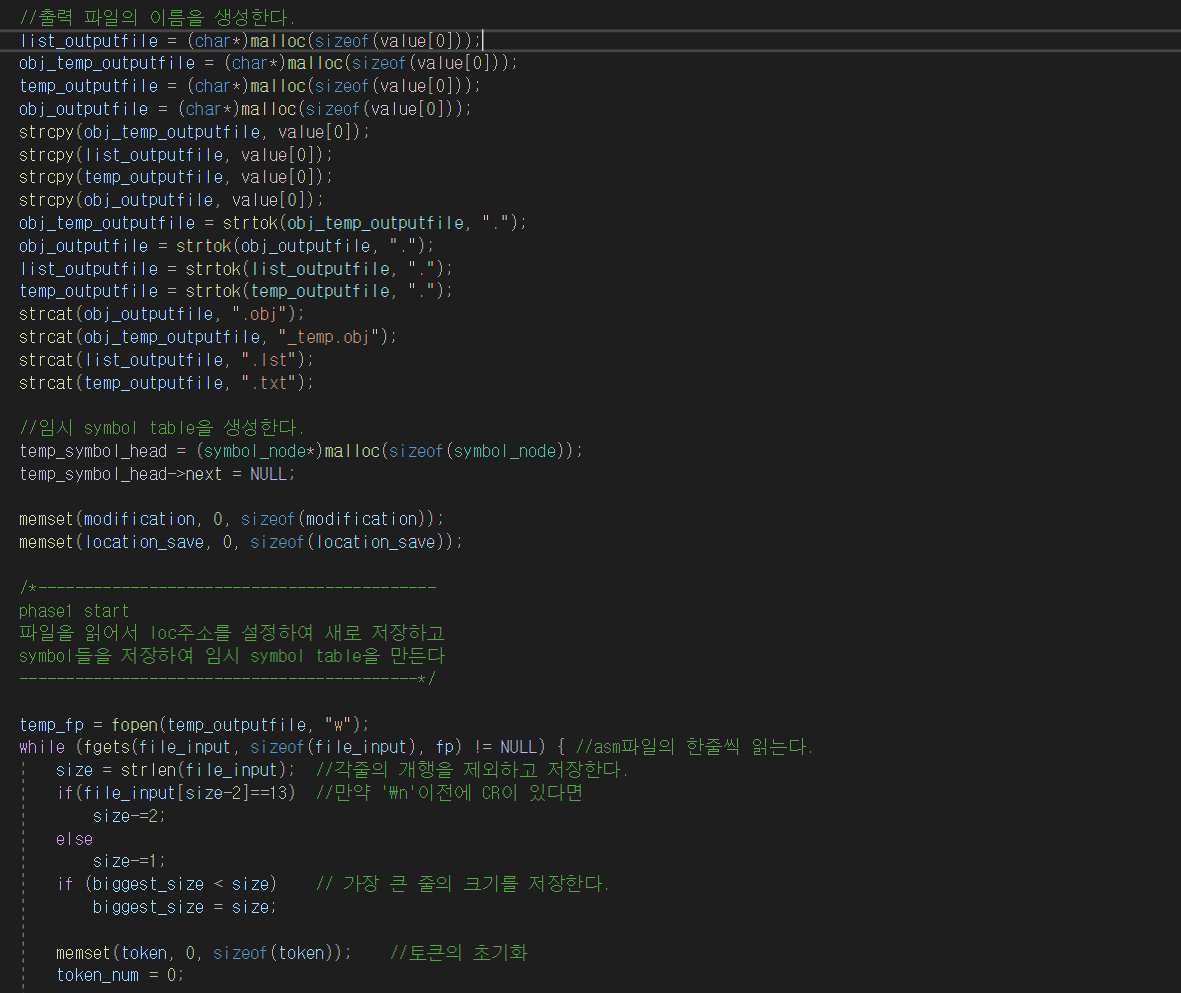
.

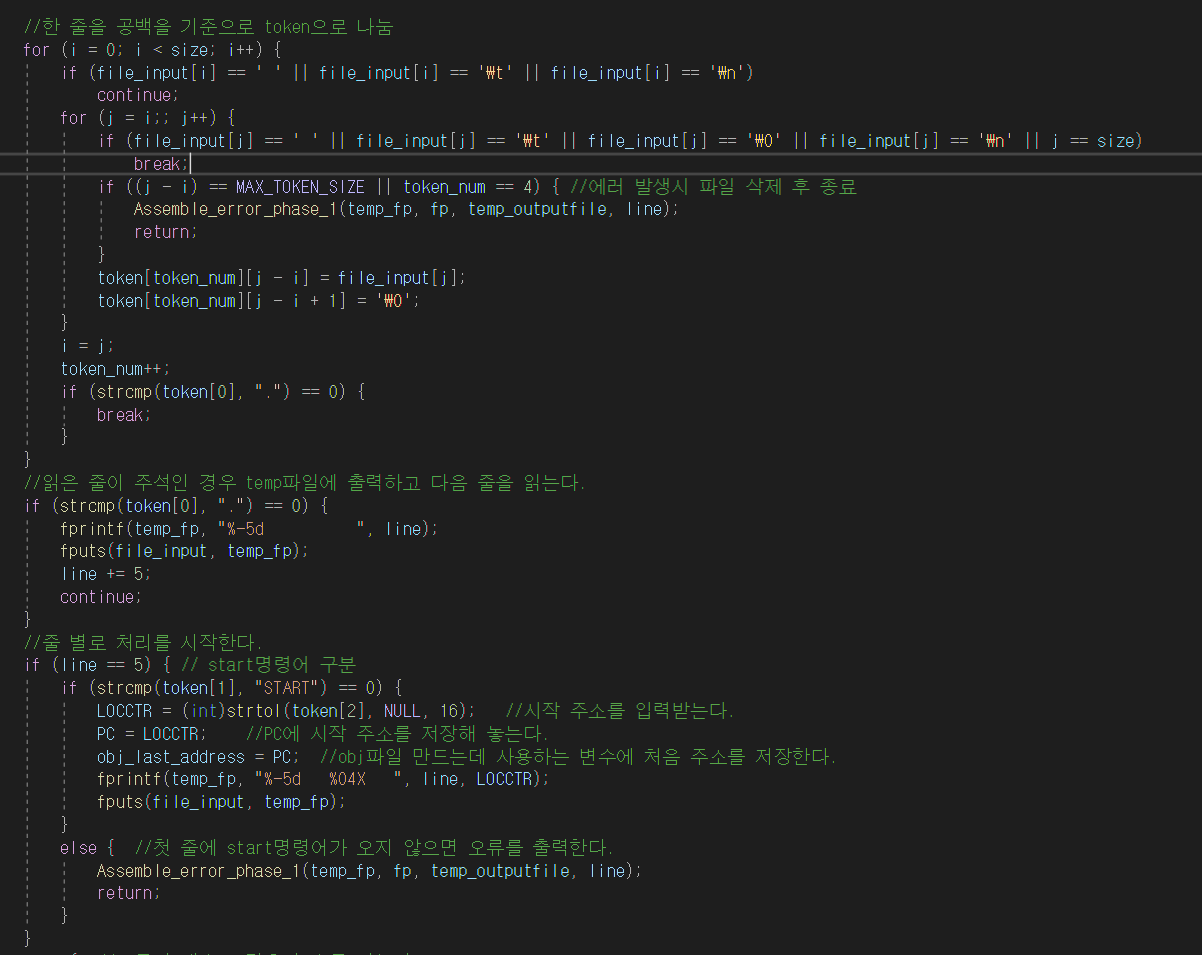
.

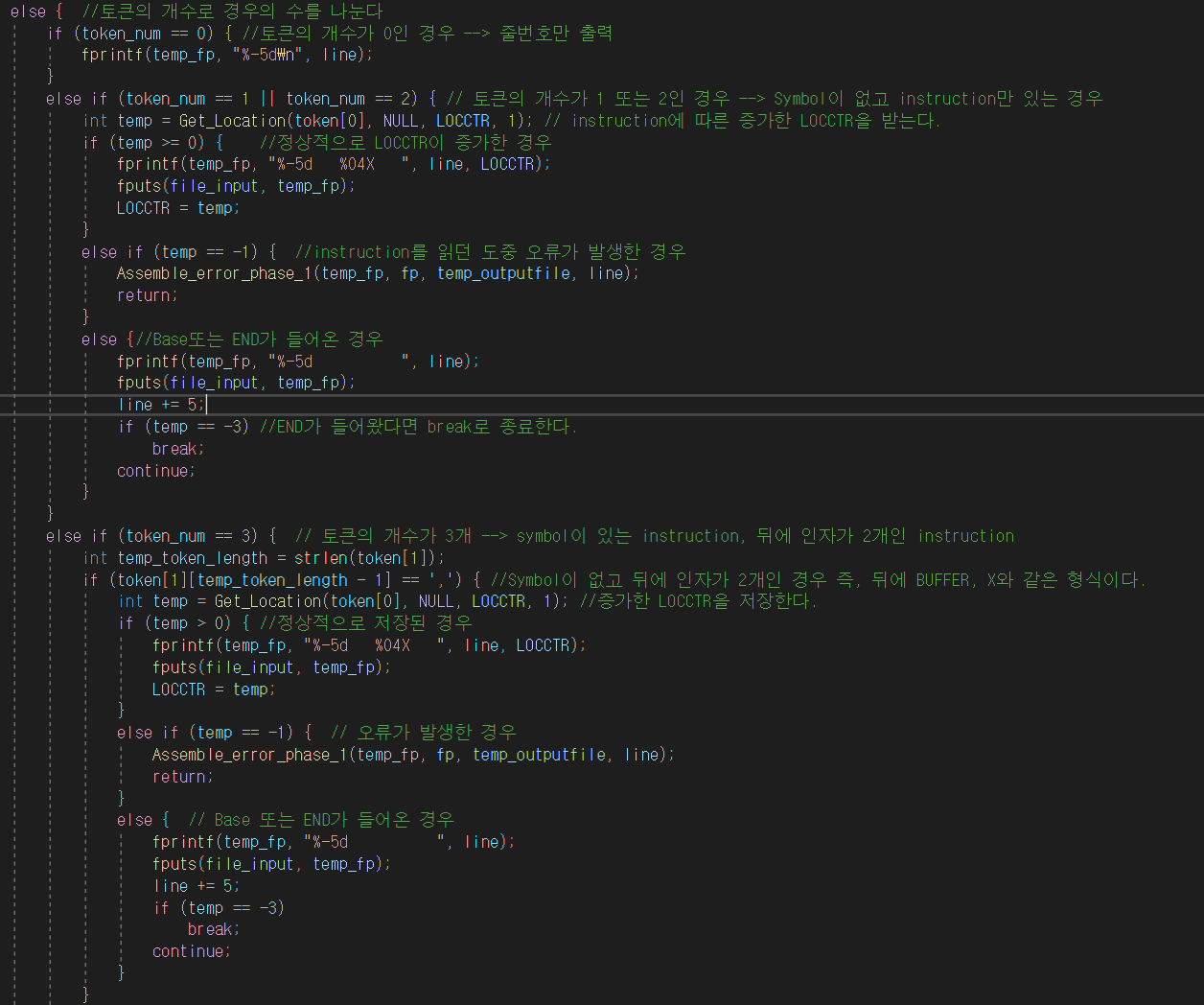
.

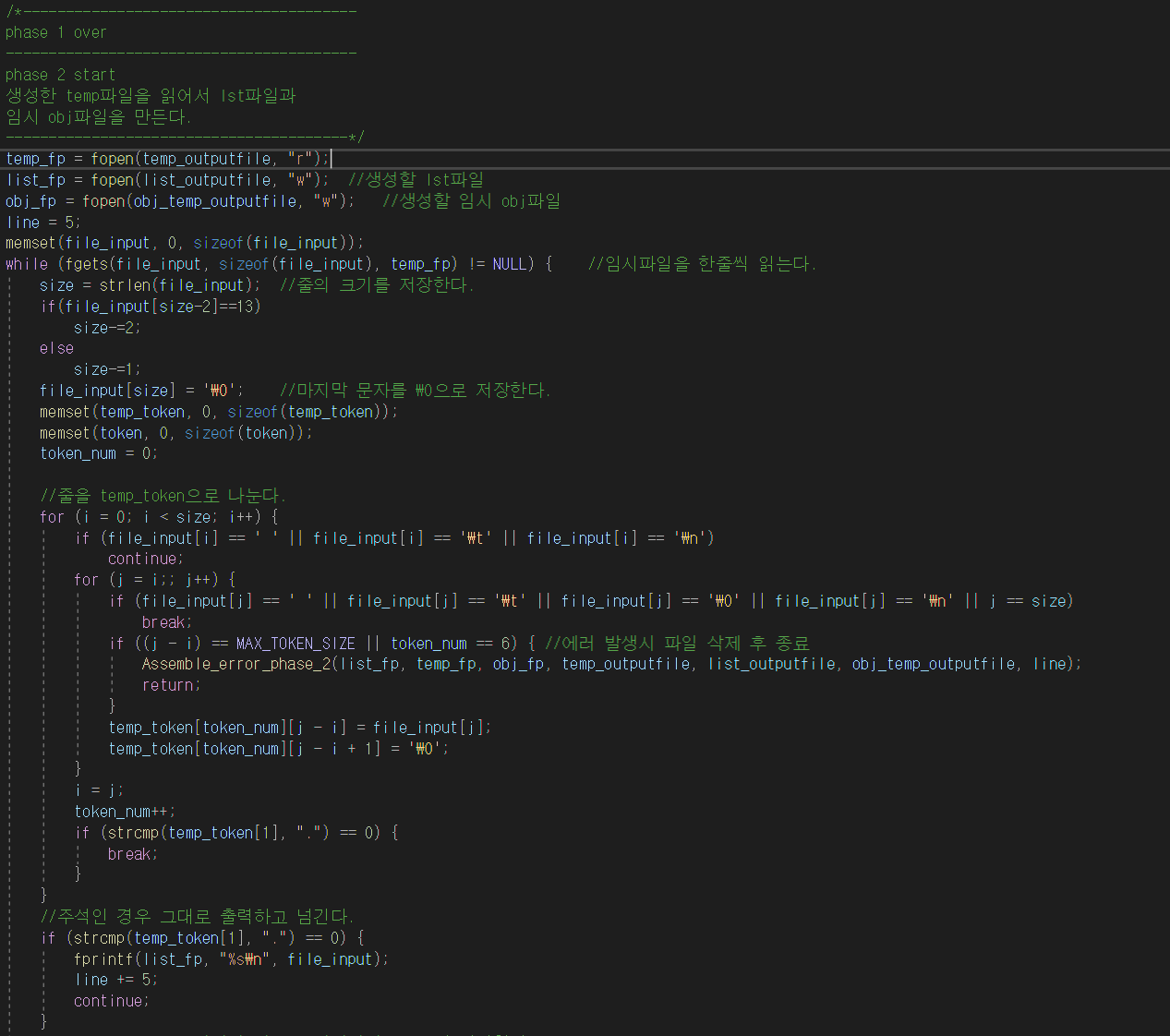
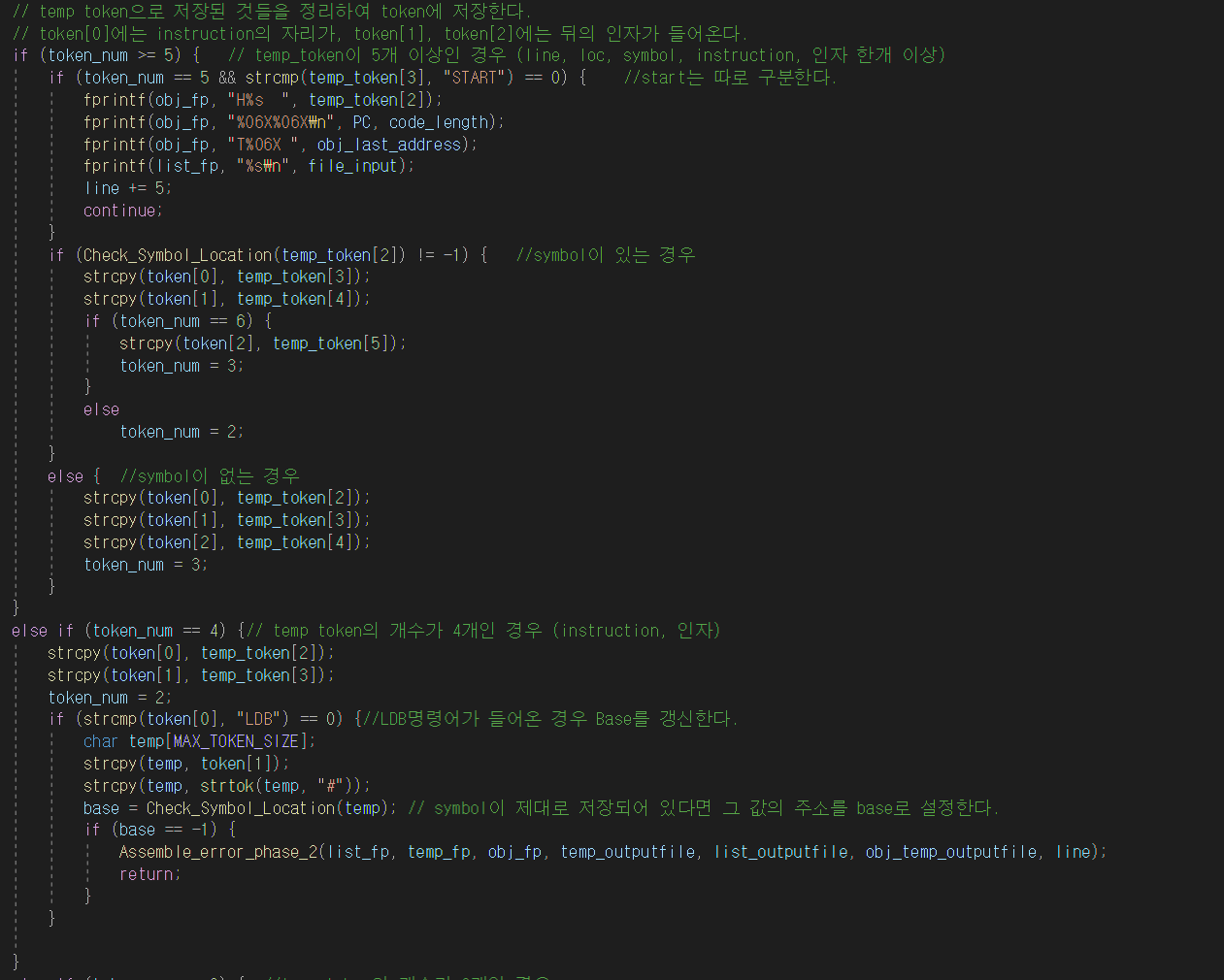


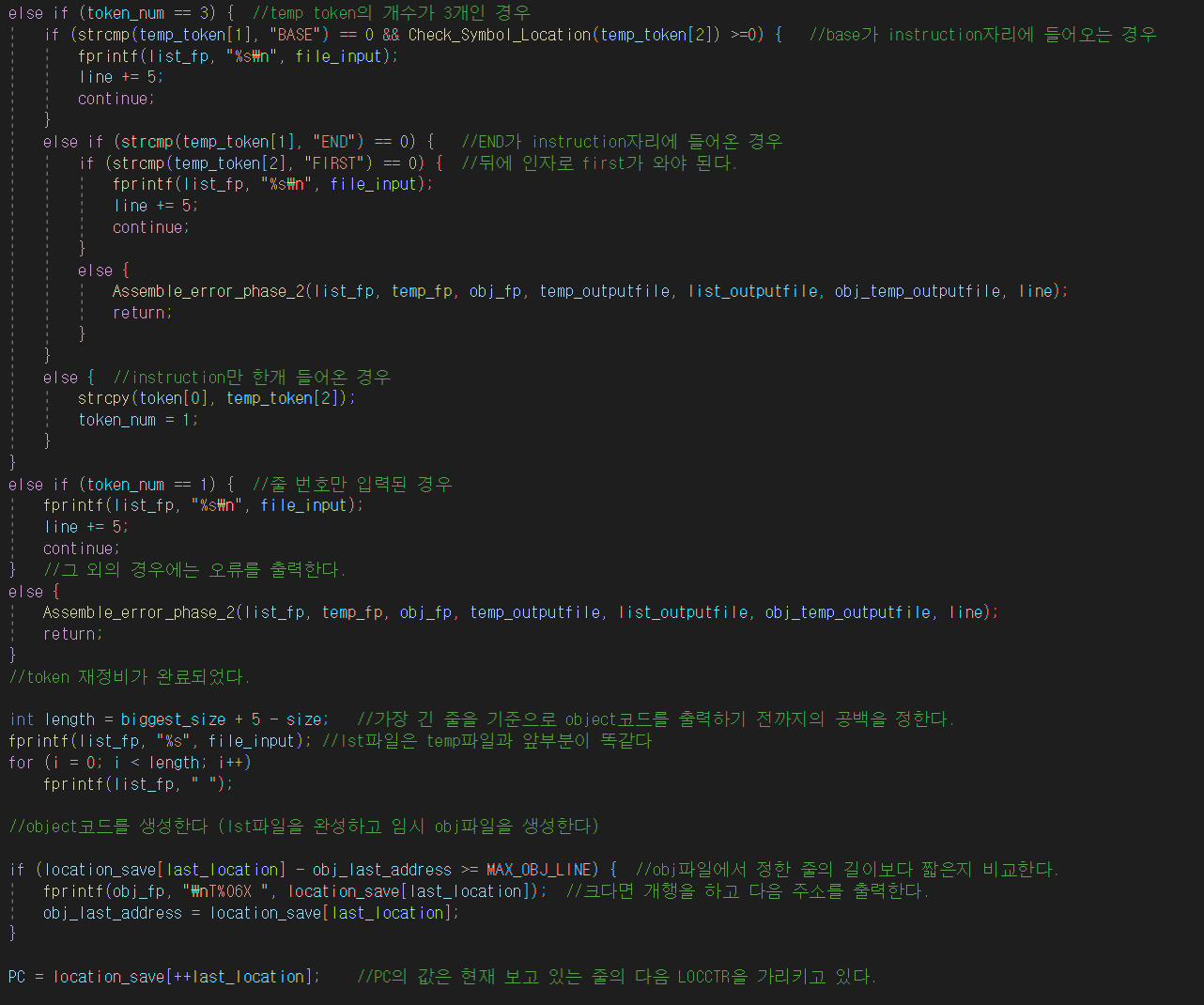


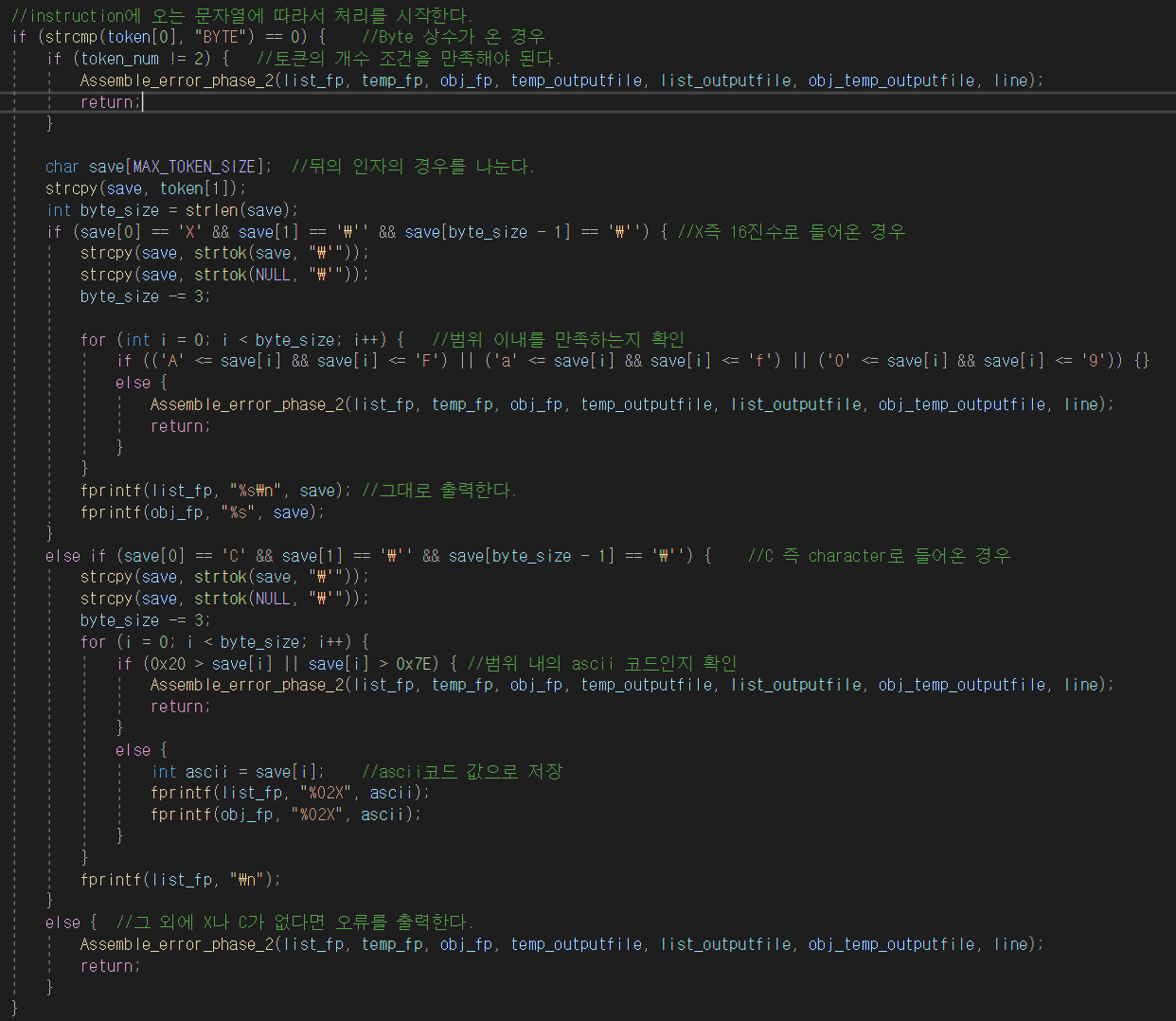


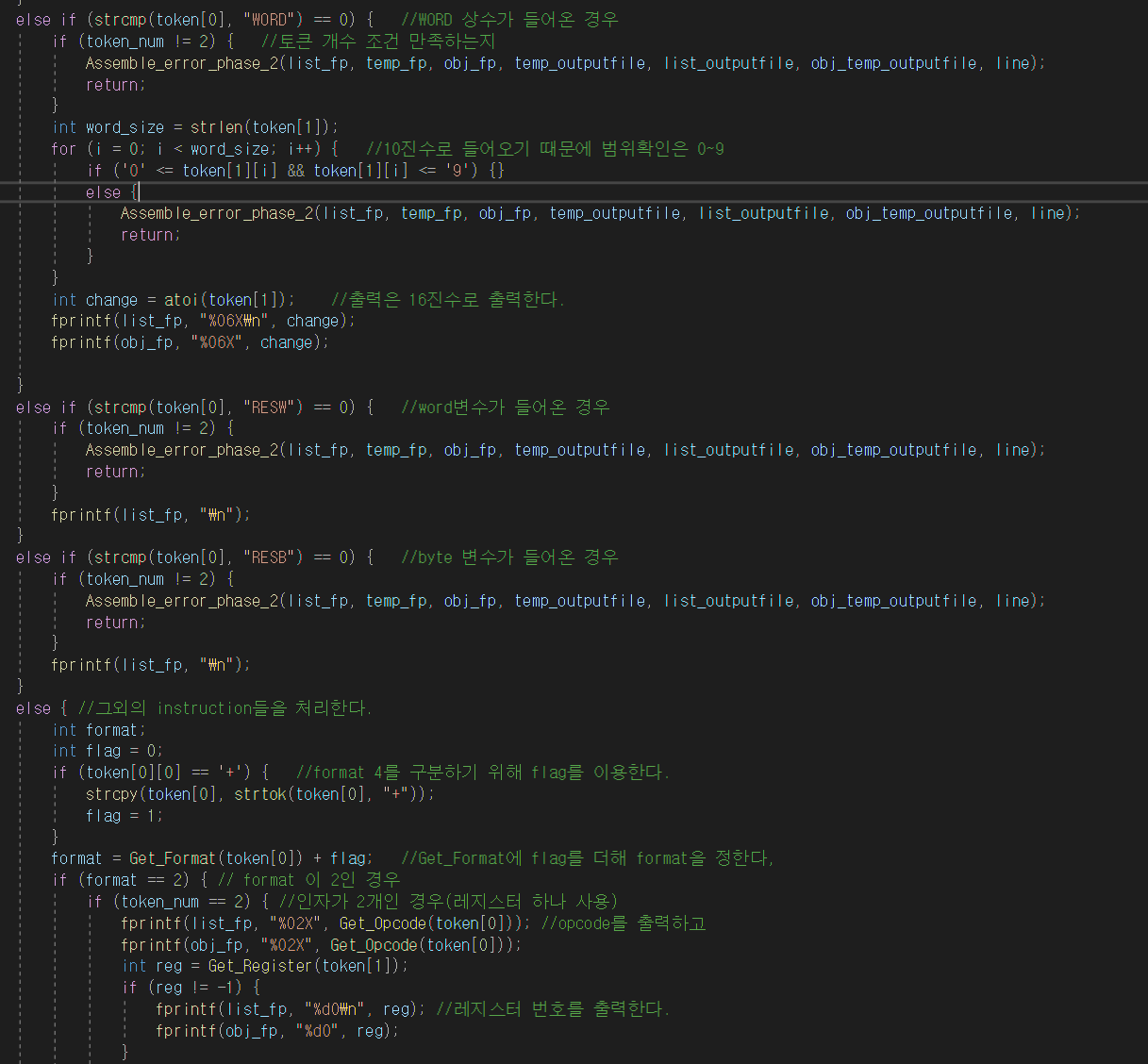
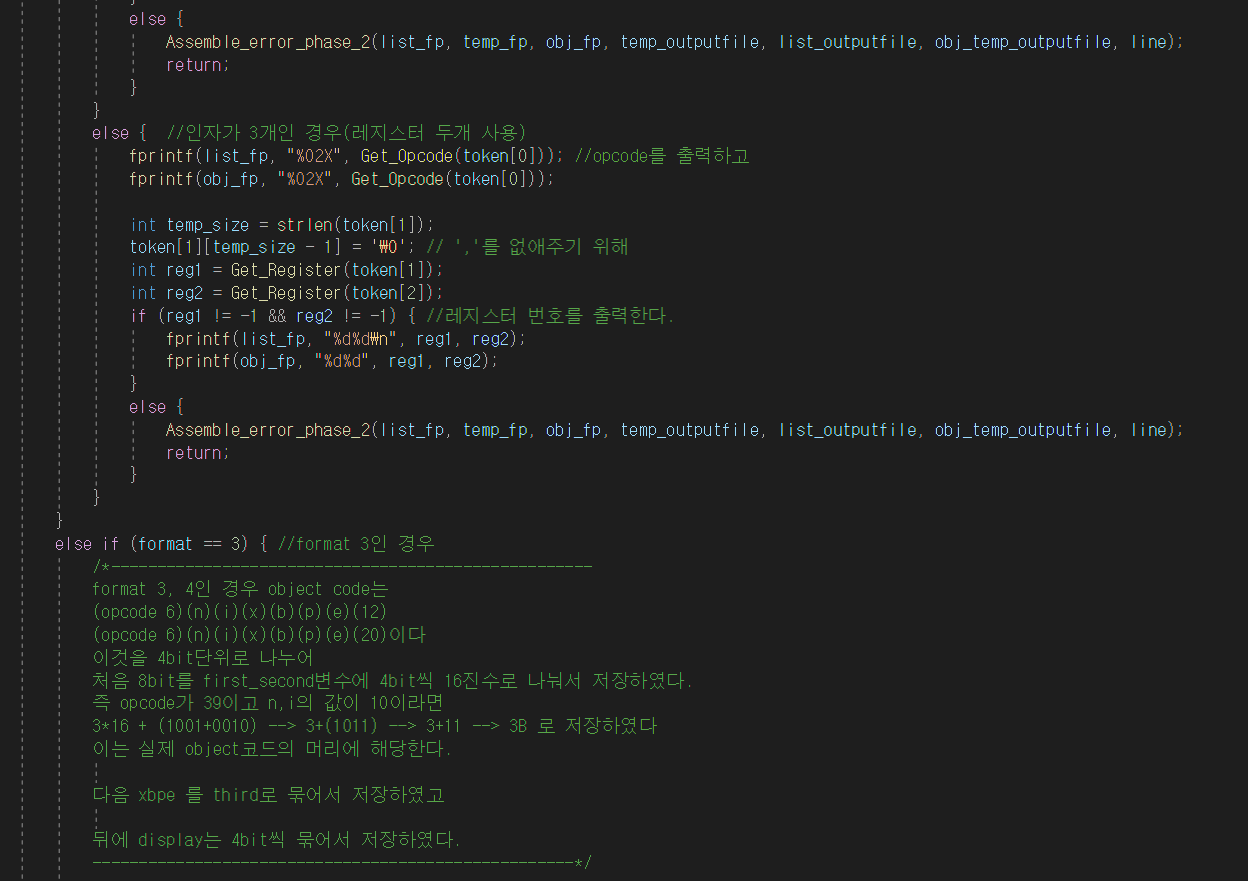


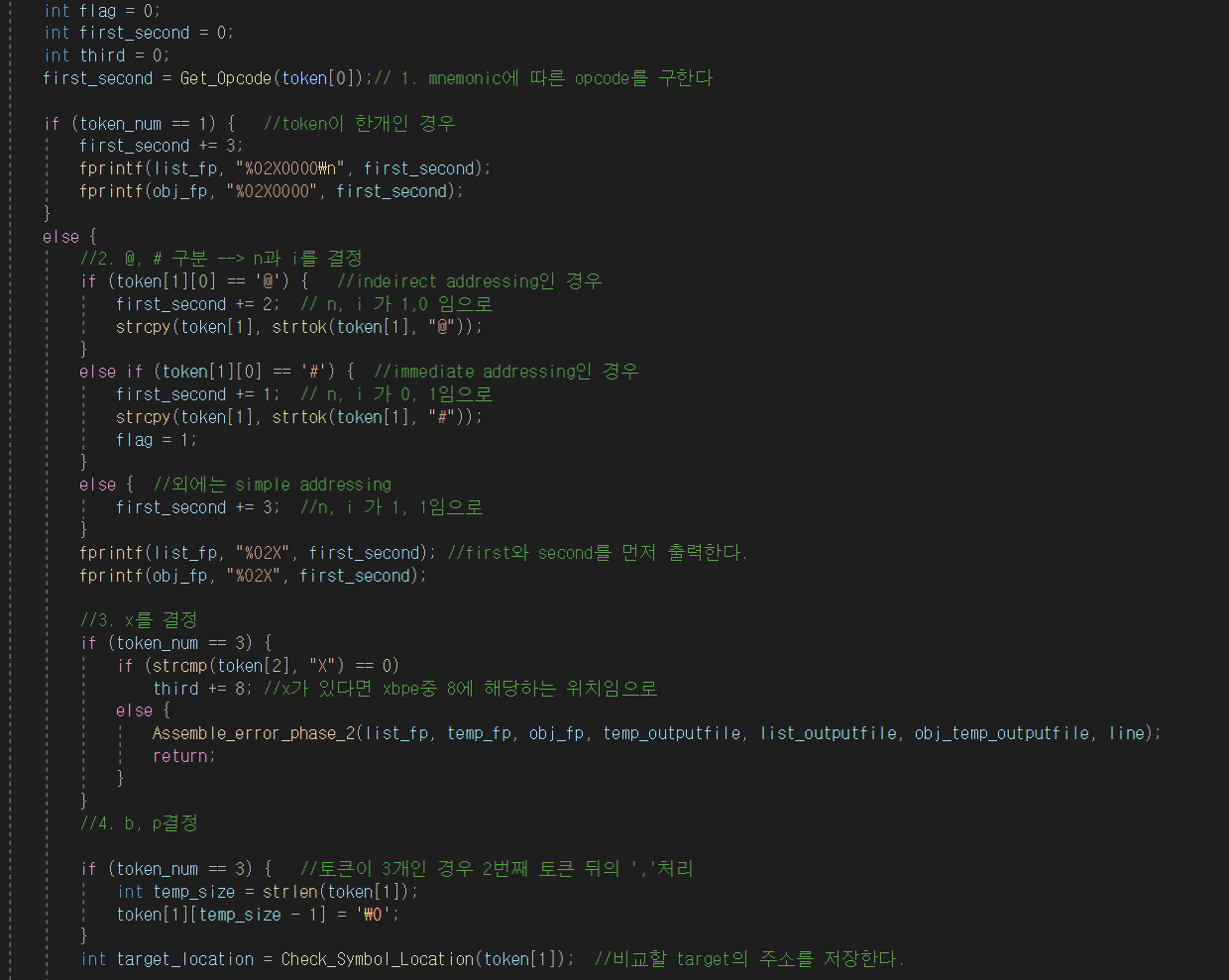
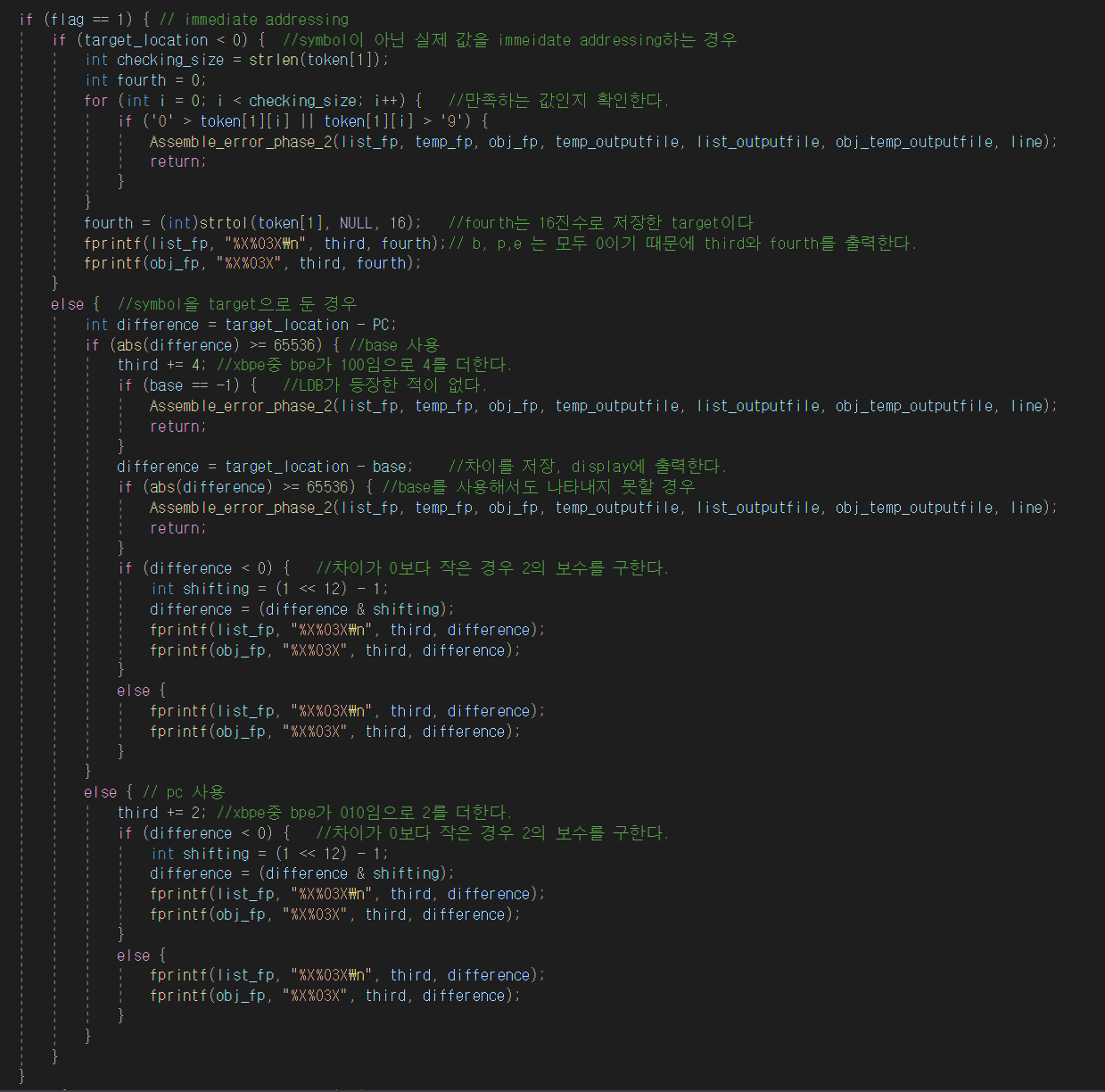
 

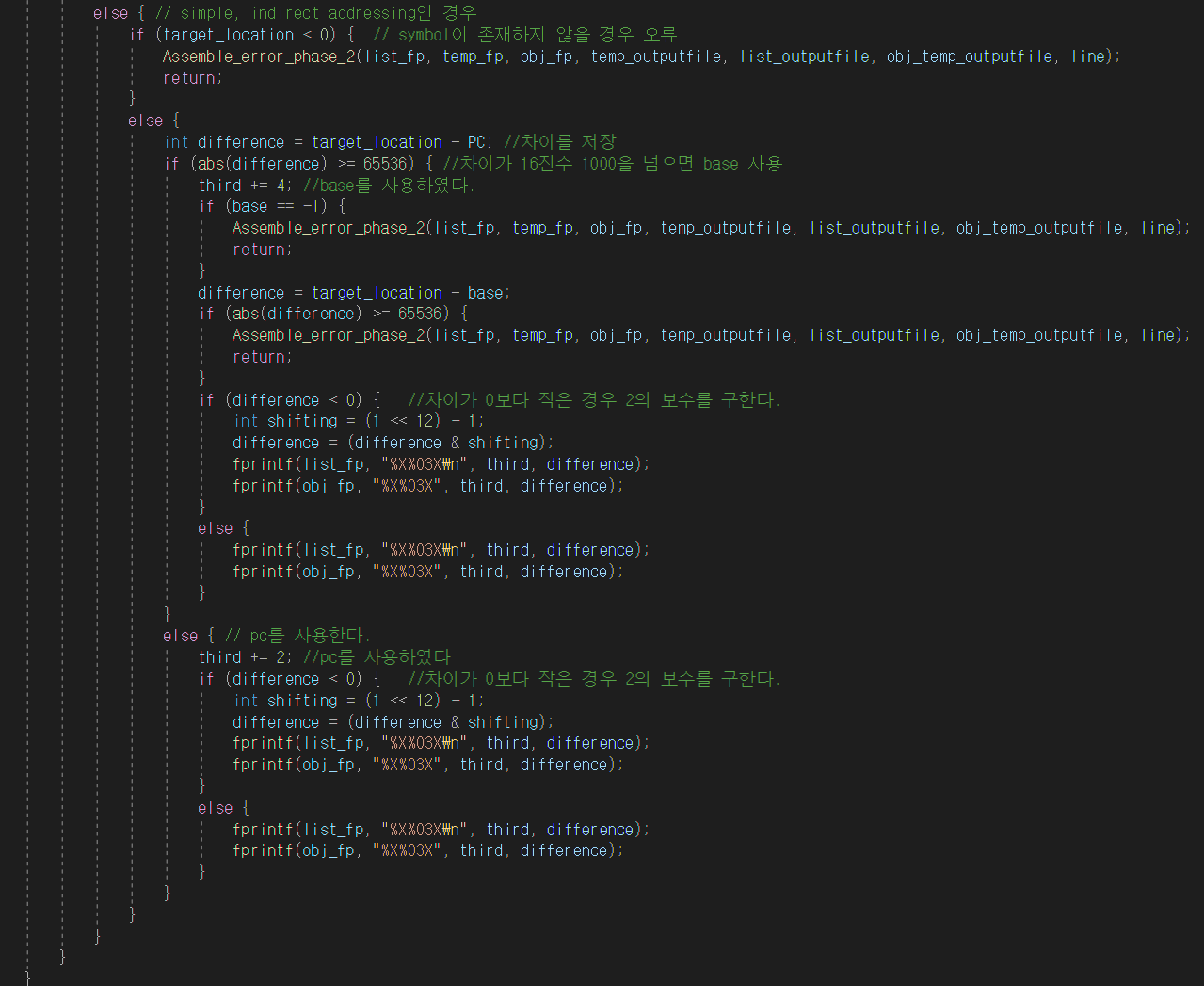
 

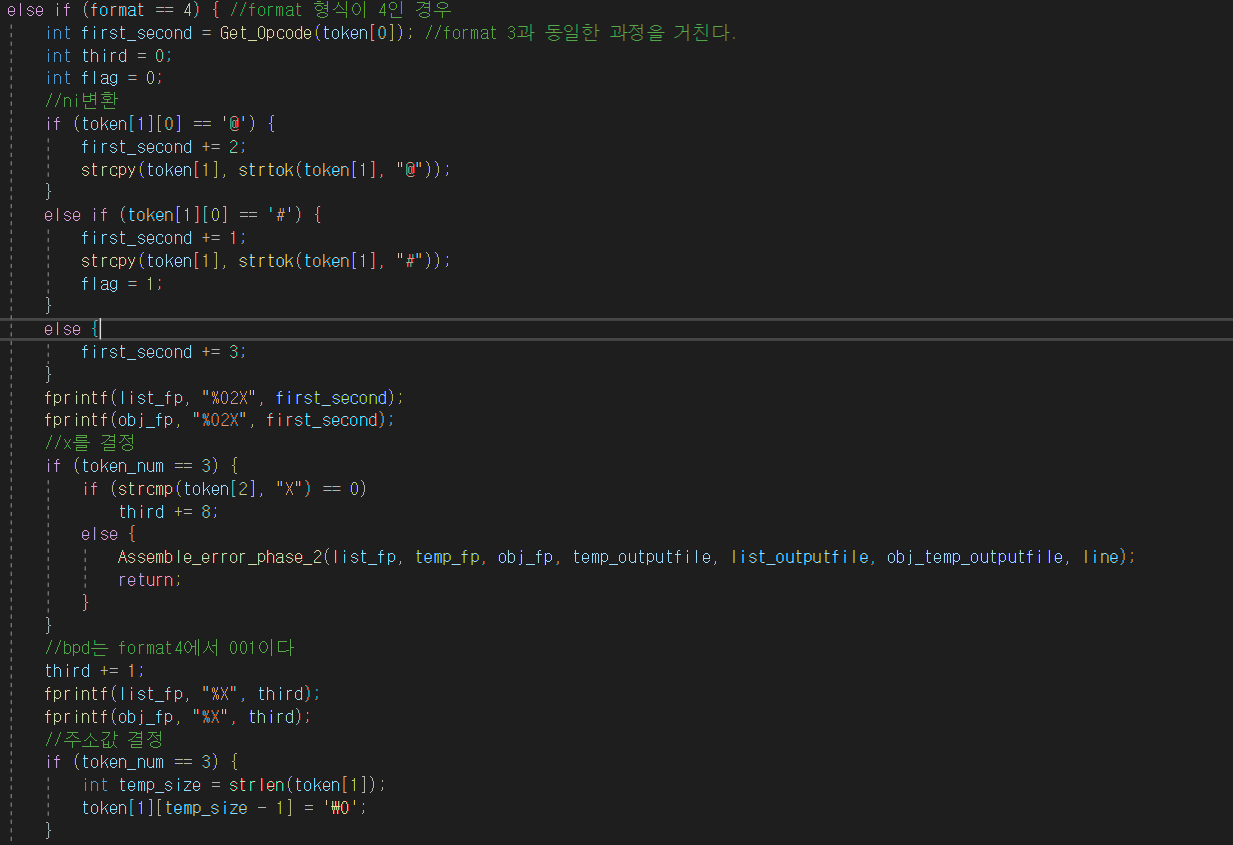




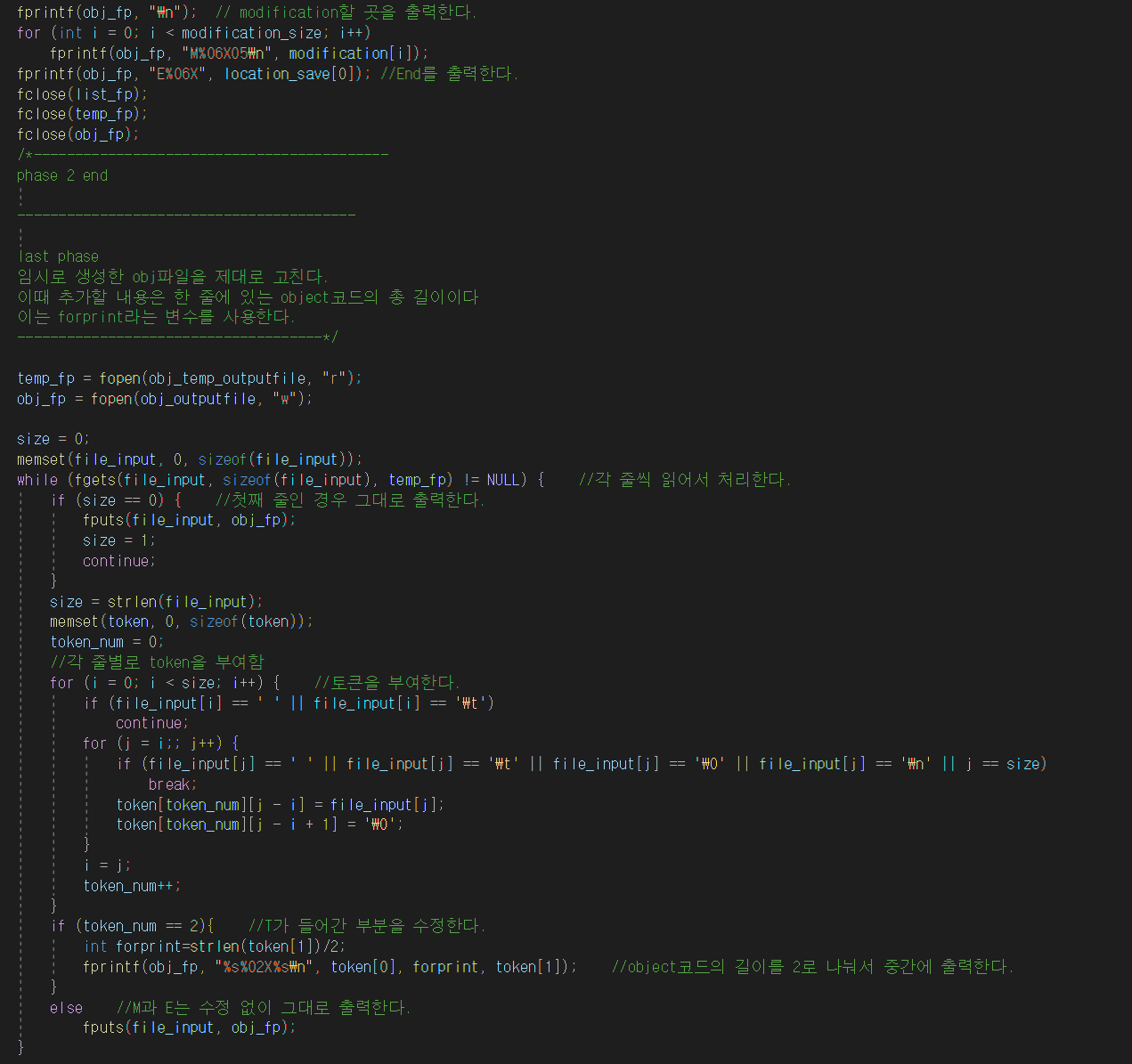
 

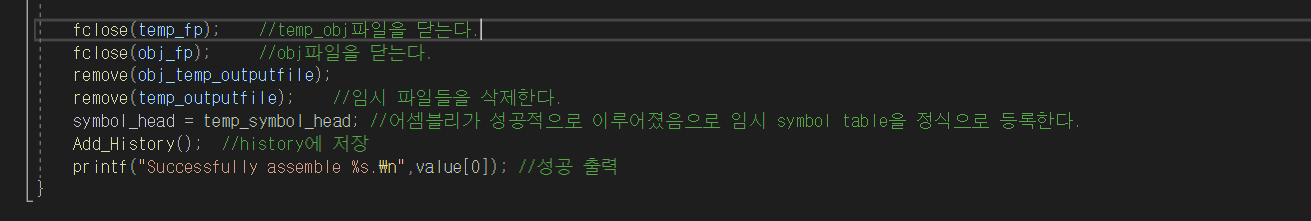
 

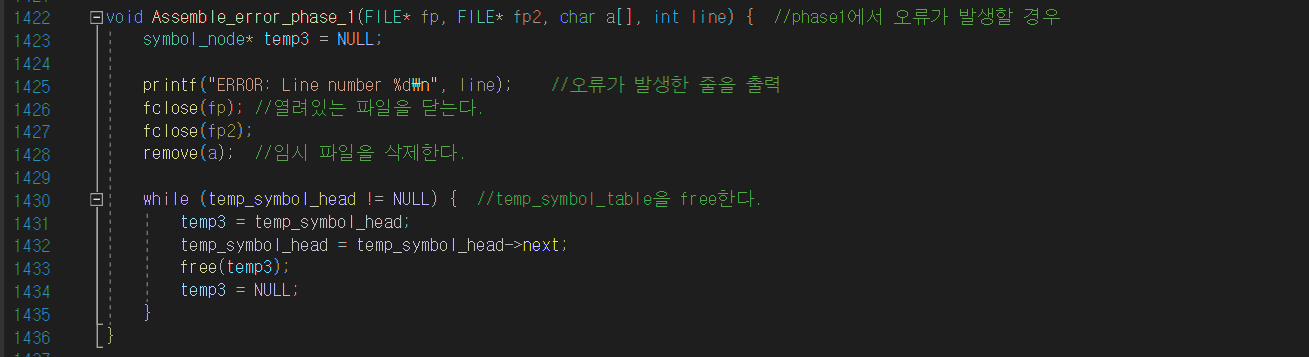


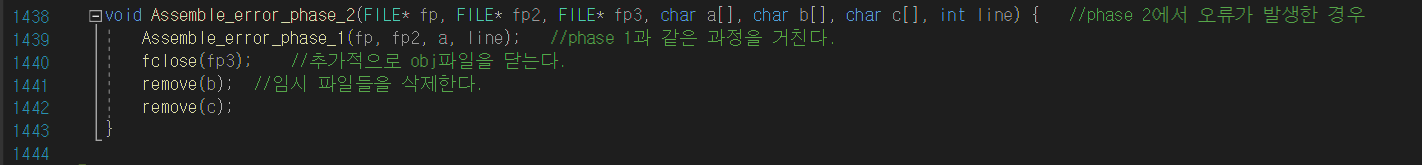


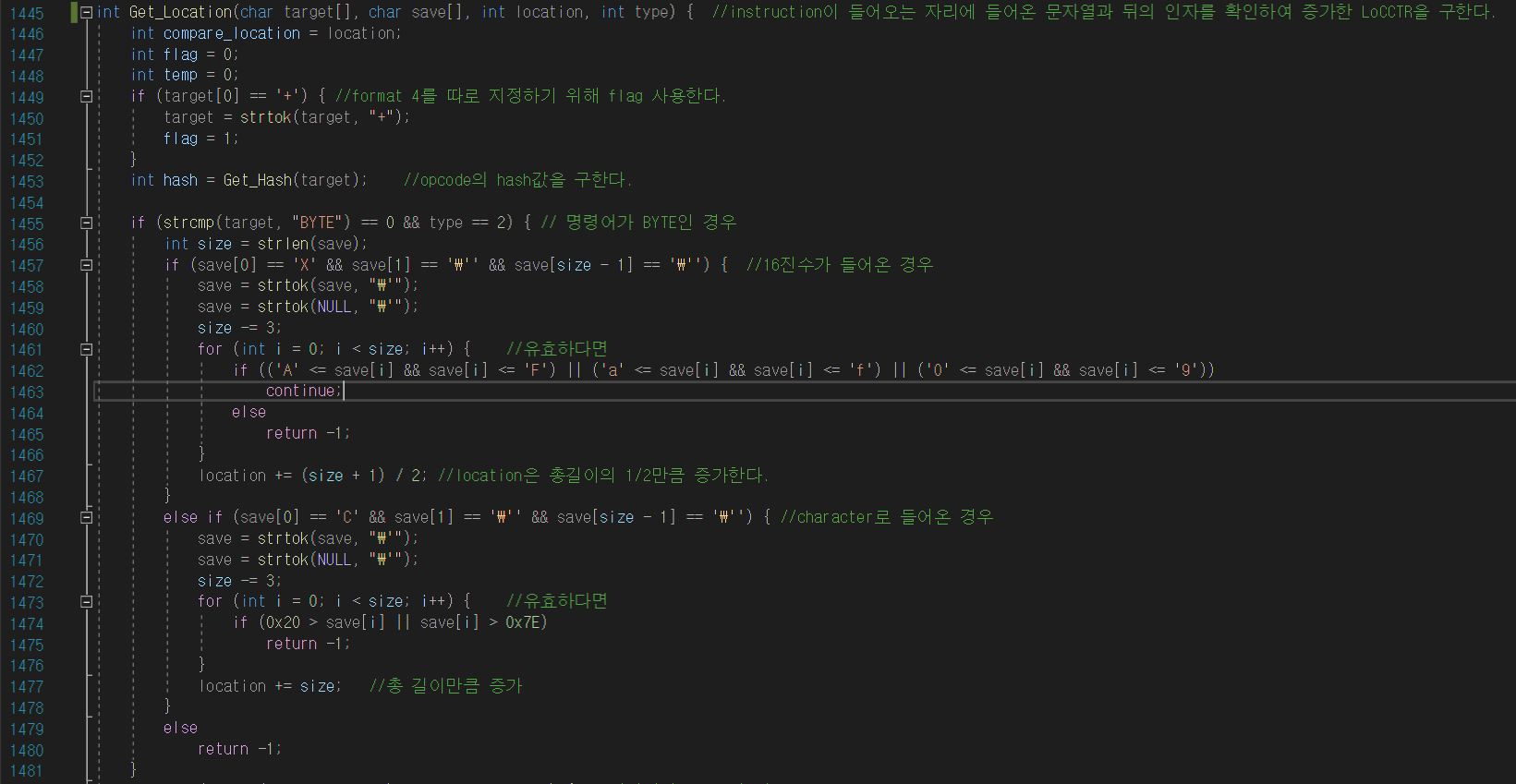


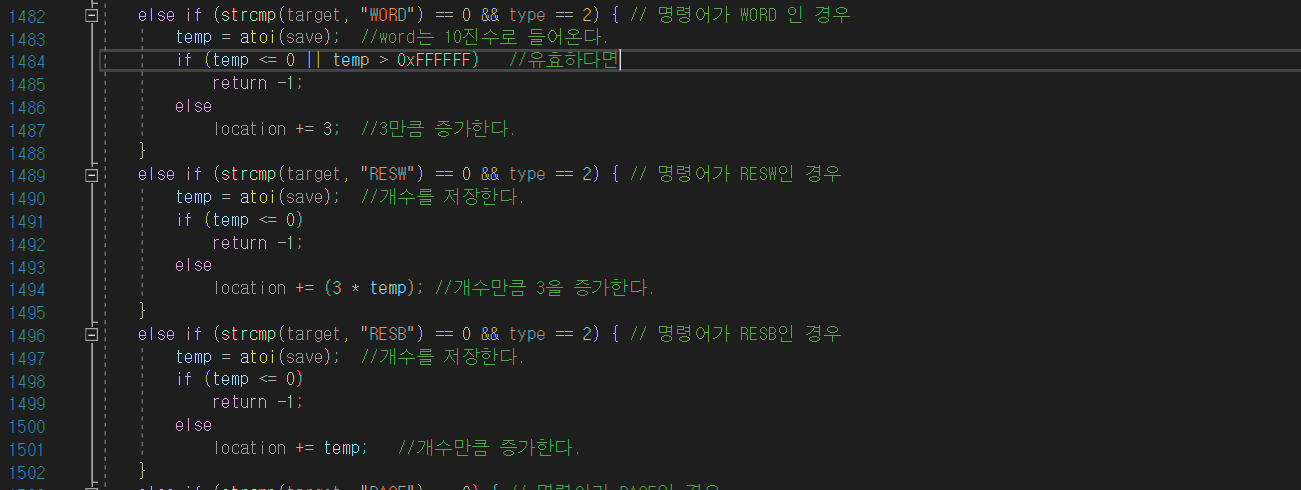


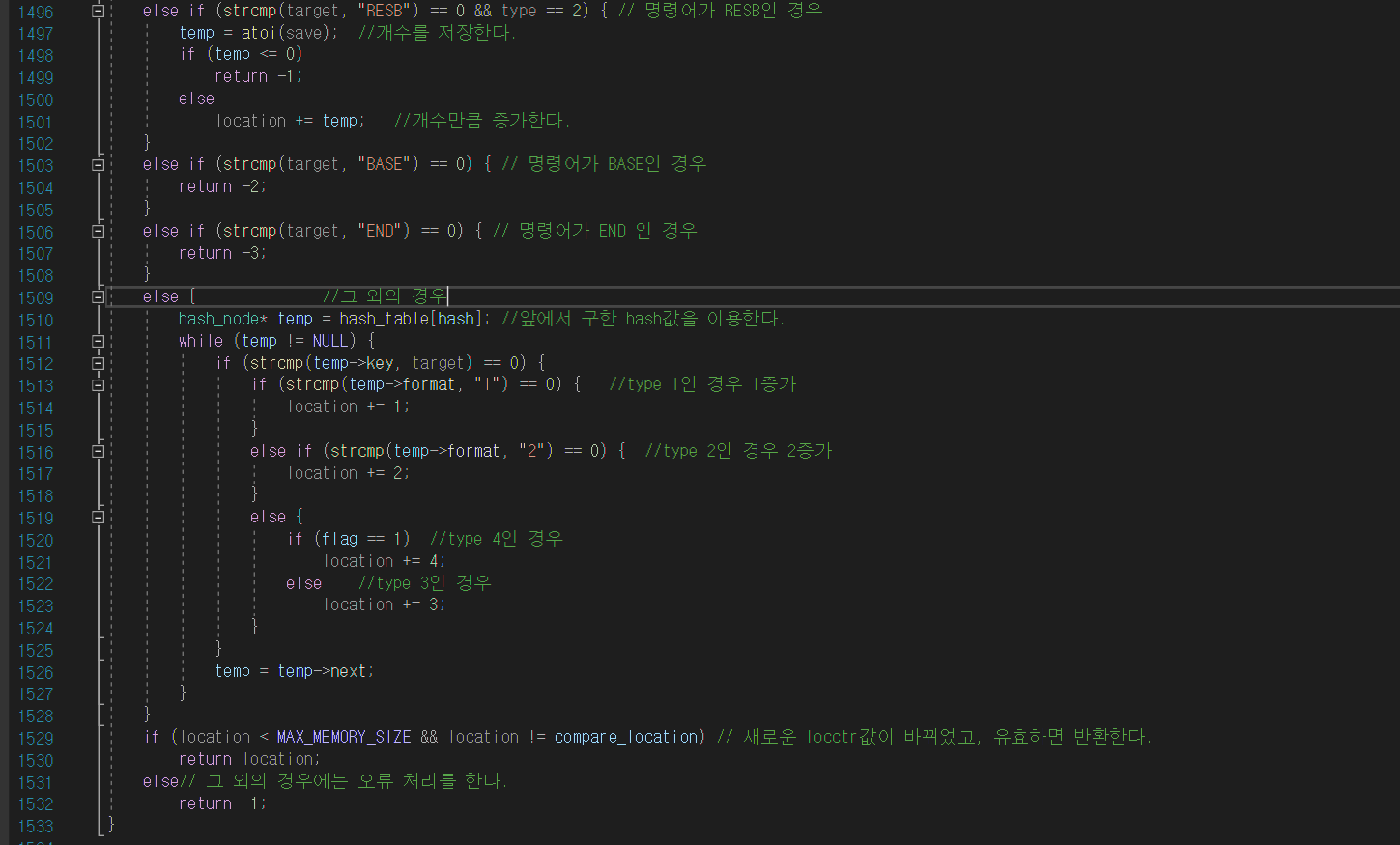


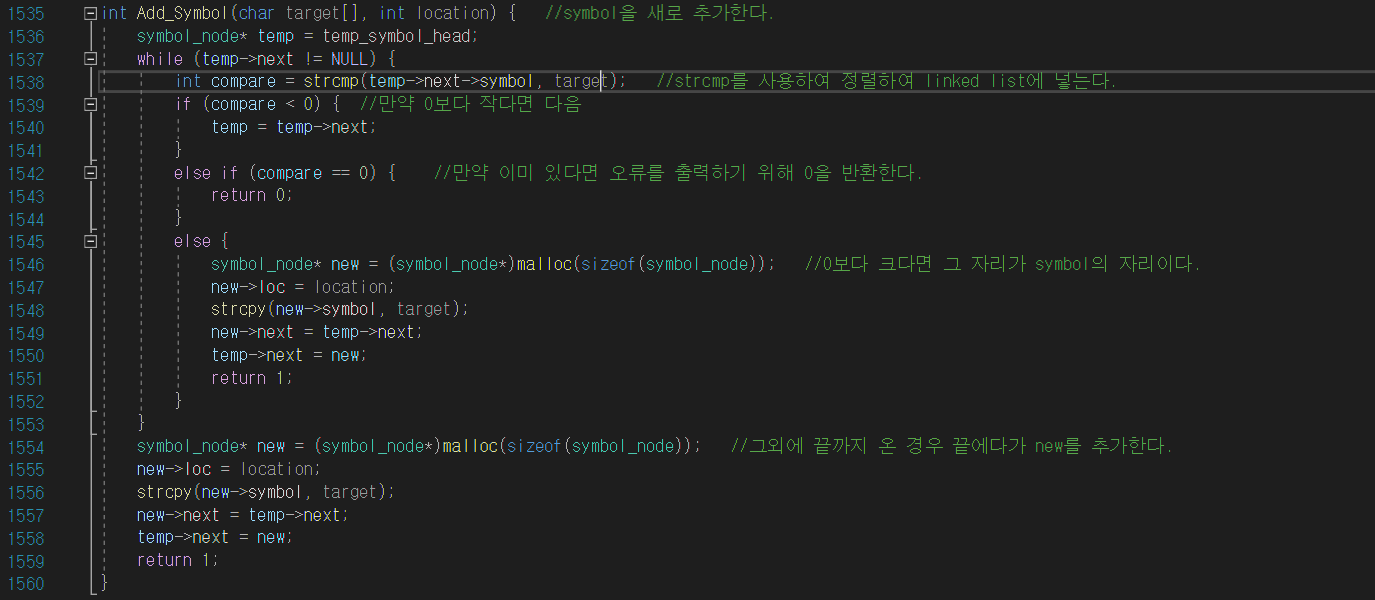


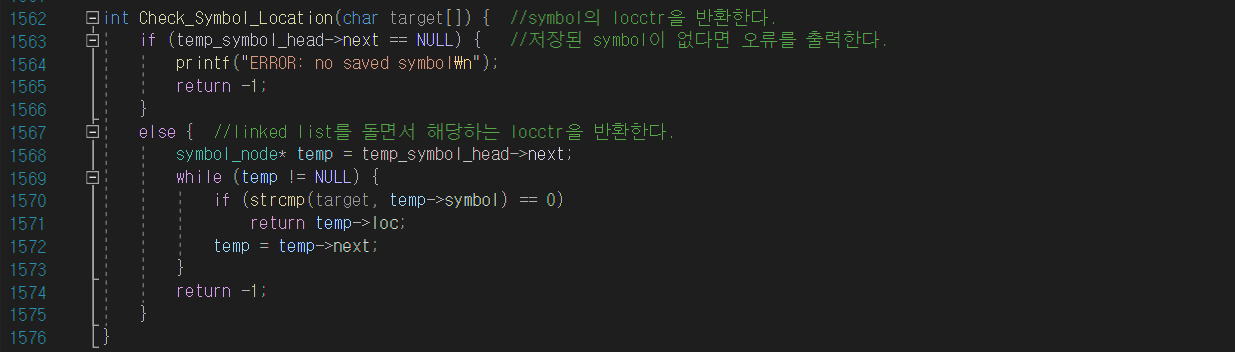


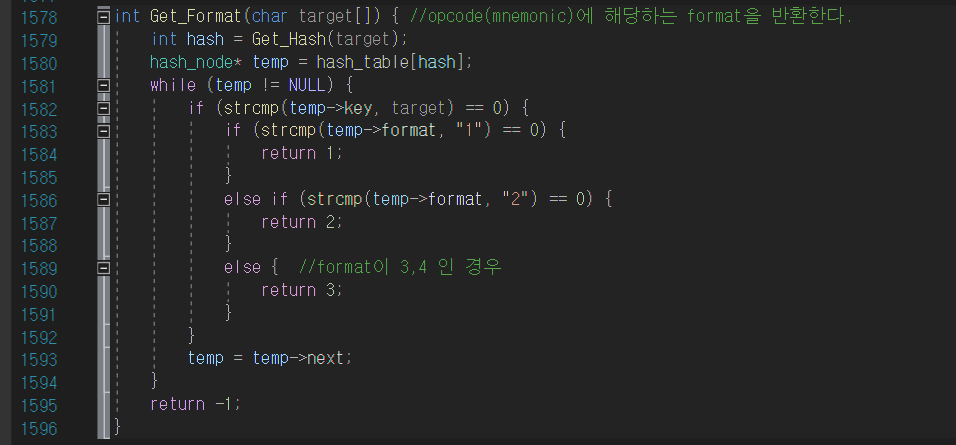
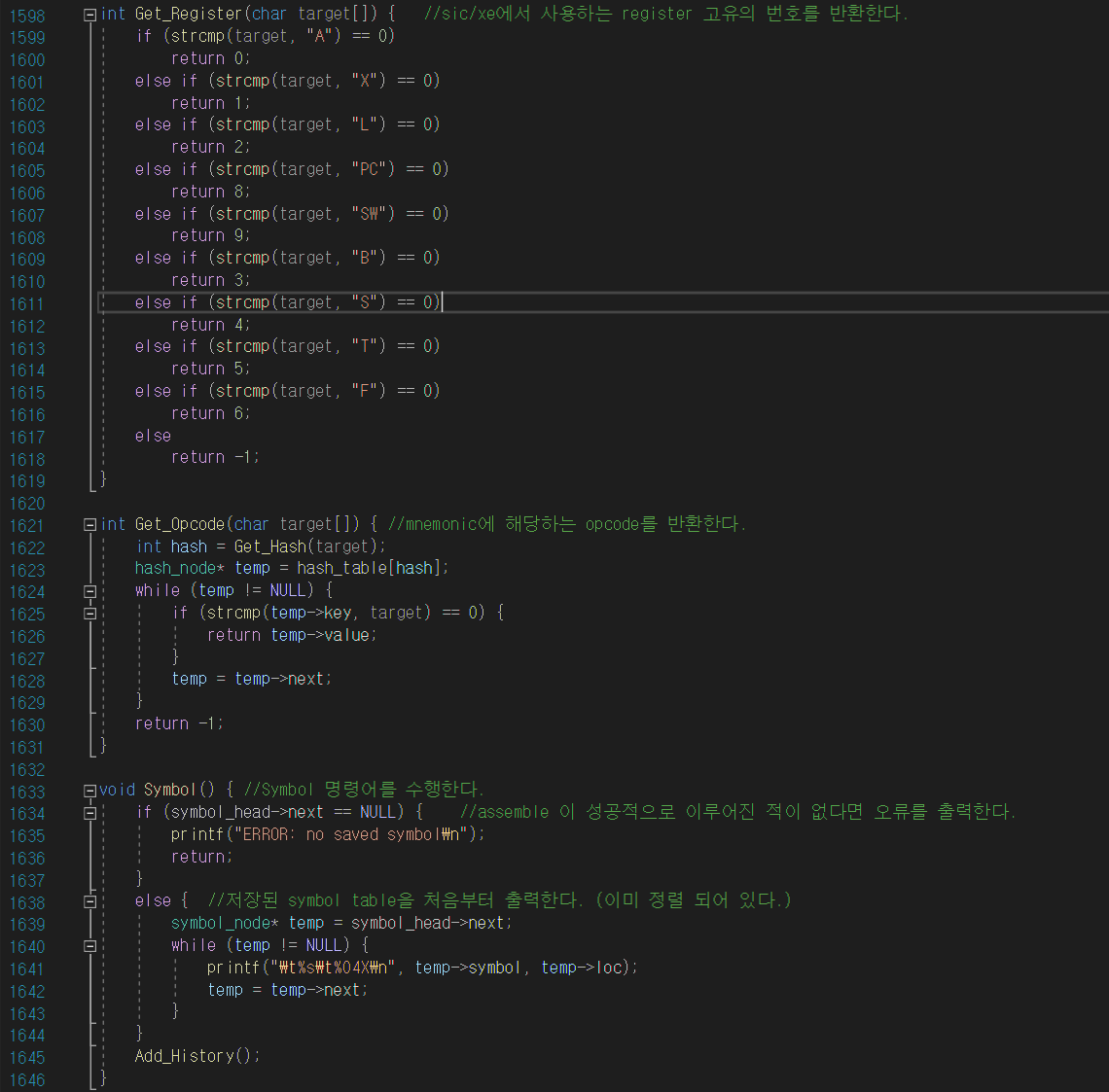




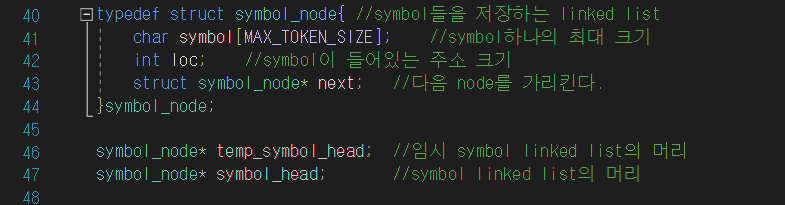






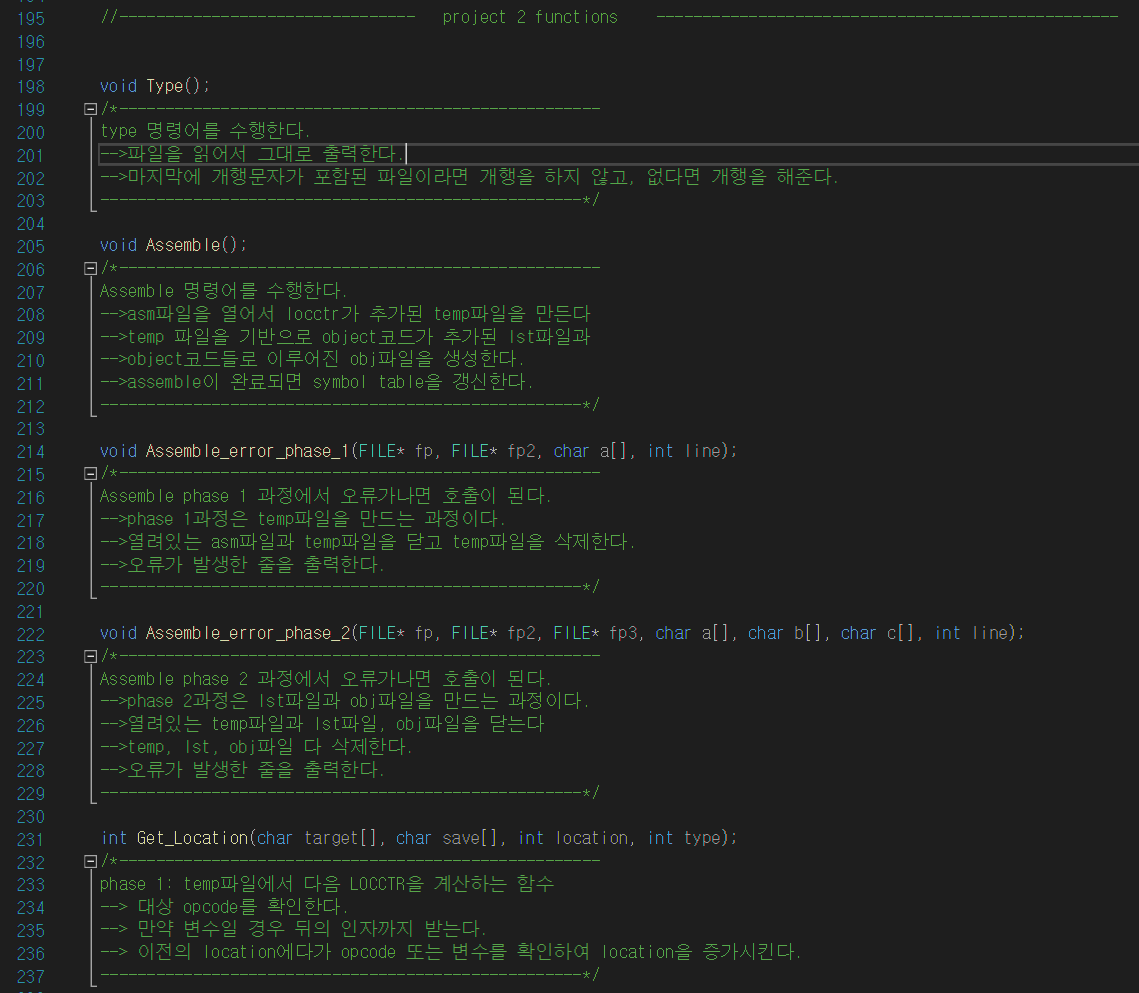
**<20161566.h>**

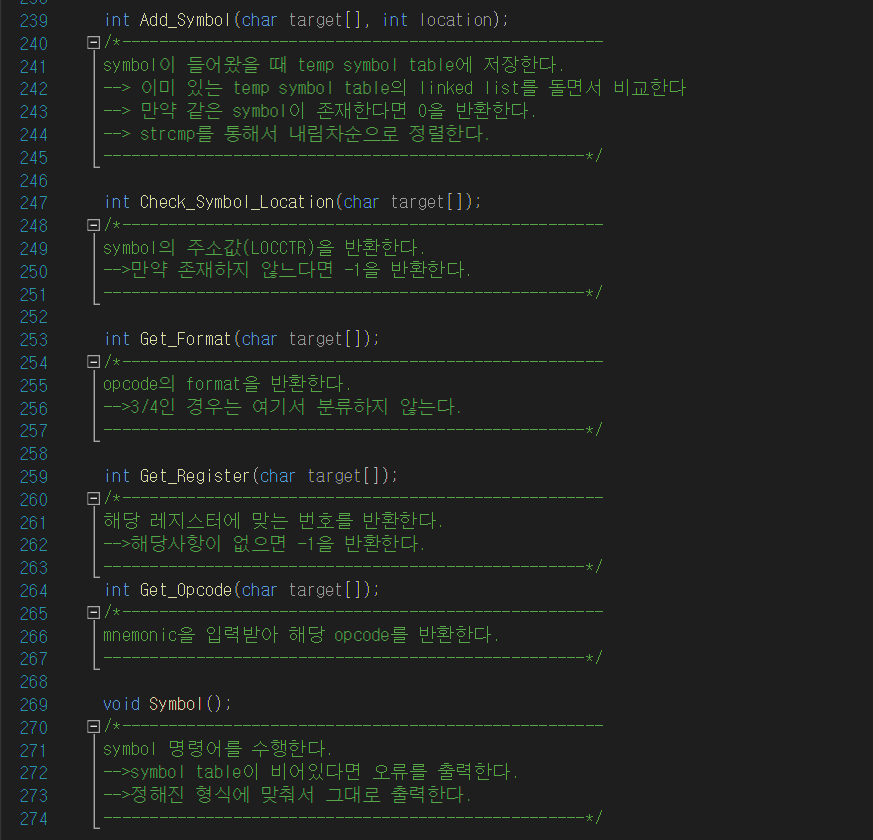


**.**

**.**

**.**





**<Makefile>**

