**과목명: 시스템프로그래밍**

**2 분반**

**<<Project #1>>**

**서강대학교 [컴퓨터공학과]**

**[20161566]**

**[권형준]**

목 차

1. **프로그램 개요**
2. **프로그램 설명**
   1. 프로그램 흐름도
   2. 흐름도 설명
3. **모듈 정의**
   1. Int main()
   2. Void Command()
   3. Int Check\_Command()
   4. Int get\_input()
   5. Void FreeMemory()
   6. Void Help()
   7. Void Quit()
   8. Void Dir()
   9. Void History()
   10. Void Add\_History()
   11. Void Dump()
   12. Void Edit()
   13. Void Fill()
   14. Void Reset()
   15. Int Check\_Address()
   16. Int Check\_Value()
   17. Int Get\_Hash()
   18. Void Add\_to\_hash\_table()
   19. Void Print\_opcodelist()
   20. Void Find\_opcode()
4. **전역 변수 정의**
   1. char \*input
   2. unsigned char\* memory
   3. value[3][10]
   4. value\_num
   5. temp\_value[4][15]
   6. temp\_value\_num
   7. address
   8. node 구조체
   9. H\_node 구조체
   10. Node\* his\_head
   11. Node\* his\_last
   12. H\_node\* hash\_table[MAX\_TABLE]
5. **코드 설명**
6. **프로그램 개요**

어셈블러, 링크, 로더들을 실행하게 될 셸(shell)과 컴파일을 통해서 만들어진 object 코드가 적재되고 실행될 메모리 공간과 mnemonic(ADD, COMP, FLOAT,etc) 을 opcode값으로 변환하는 OPCODE 테이블과 관련 명령어들을 구현하는 프로그램이다.

1. **프로그램 설명**
   1. **프로그램 흐름도**

**텍스트, 지도이(가) 표시된 사진

자동 생성된 설명**

* 1. **흐름도 설명**

흐름도를 간단히 설명하자면, 처음 프로그램이 시작되고 opcode.txt를 열어 hashtable을 만든다. 이에 성공하면 input을 받는 단계를 거치고, input의 유효성을 판단하게 된다. 이때 Check\_Command함수를 통하여 명령어가 유효한지 먼저 판단하고, 명령어에 따른 인자의 개수가 유효한지 판단한다. 만약 뒤의 인자가 주소 값이나 메모리 값을 의미하는 인자가 들어오면, Check\_Value와 Check\_Address함수를 통하여 유효한 주소 값, 메모리 값인지 확인한다. 위의 조건을 모두 만족하면 Check\_Command함수는 0이 아닌 값을 Command에 반환하게 되고, Command에서 수행할 함수들을 호출하게 된다. 이때 오류가 발생하지 않는 함수들은 실행된 뒤 Add\_History를 통해서 history에 추가된다. (단, history()명령어는 먼저 add\_history가 진행된다.) 또한 오류가 발생하는 함수들은 오류가 발생하면 Error를 출력한다. 그리고 다시 get\_input 함수를 호출하며 반복된다. Quit이 호출되면 메모리를 free하는 FreeMemory함수를 호출하고 종료한다.

1. **모듈 정의**
   1. **Int main()**

변수 설명

🡪Int temp\_opcode: opcode.txt에서 읽어온 opcode를 임시 저장한다.

🡪Char temp\_key[MAX\_KEY]: opcode.txt에서 읽어온 mnemonic을 임시 저장한다.

🡪Char temp\_format[MAX\_FORMAT]: opcode.txt에서 읽어온 format을 임시 저장한다.

함수 설명

프로그램이 시작되면서 초기화되고, 메모리가 할당되야 되는 것들을 할당하면서 시작된다. Input, memory, history를 생성하고 address를 초기화한다. 다음 opcode.txt를 열어서 읽은 후 get\_hash와 add\_to\_hash\_table을 호출해 hash\_table을 생성한다. 다음 반복문을 통해서 quit이라는 명령어가 들어오기 전까지 input을 받고 일을 수행하고 input을 초기화하는 과정을 반복한다.

* 1. **Void Command()**

변수 설명

🡪int a: Check\_Command()함수를 호출하고 반환 받은 값이다. 이 값은 switch문에서 사용되어 Command()함수가 어떤 함수를 호출할지 정해준다.

함수 설명

Command()함수는 input이 있을 때, 즉 get\_input()의 반환 값이 1일 때 호출이 된다. 이 함수는 Check\_Command()를 통해서 구분된 명령어에 따른 함수를 호출한다. 간단히 switch문을 사용하여 10개의 명령어에 따른 함수를 호출하였고, 그 외의 경우에는 아무것도 하지 않는다.

* 1. **Int Check\_Command()**

변수 설명

🡪int size: strlen을 사용하여 input값의 총 길이를 저장한다.

🡪int i, j: for문을 사용하기 위해 선언

함수 설명

우선 input값은 하나의 긴 문자열임으로, 이 문자열을 공백과 ‘,’를 기준으로 파싱해야 한다. 그러기 위해 2중 for문을 사용하여 input을 돌면서 차례대로 temp\_value에 각 토큰을 저장하였다. 만약 저장 중, 토큰의 개수가 너무 많거나 하나의 토큰의 길이가 너무 긴 경우에는 오류를 출력하고 0을 반환하였다. 이 과정이 끝나면 temp\_value[]에는 각 토큰이 저장되게 된다. 만약 3번째 인자까지 존재하는 경우 2번째 인자 뒤에는 ‘,’이 붙어있음으로 이 ‘,’를 없애고 ‘\0’을 넣어주는 것으로 이 과정이 끝난다. 이후 가장 첫번째 토큰에는 명령어가 저장되어 있어야 한다. If, else if구문을 사용하여 각각 명령어에 맞는 처리를 한다. 첫 토큰이 명령어에 해당하는 토큰이 아니라면 0을 반환한다. 이후 각 명령어에 따른 허용가능한 인자의 개수를 확인한다. 예를 들어서 help는 토큰이 오직 help 하나만 있어야 된다. 만약 토큰의 수가 이에 벗어난다면 오류를 출력하고 0을 반환한다. 명령어도 맞고, 인자의 수도 충족하였다면, 인자로 오는 값이 유효한 값인지 확인해야 한다. dump같은 경우 뒤에 주소 값이 올 수 있다. edit같은 경우에는 주소 값과, 메모리 값이 들어온다. 명령어와 인자의 수를 조합하여 인자가 메모리 값인지 주소 값인지 mneomnic인지 구분을 한다. 만약 주소 값인 경우 Check\_Address를 통해서 이 주소가 유효한지도 체크한다. 만약 메모리 값이라면 Check\_Address를 통해서 이 값이 메모리 값에 유효한지 체크한다. mnemonic이라면 해당 토큰이 영문 대문자로 이루어져 있는지 확인한다. 위의 예외를 다 거르고 만족한 입력은 temp\_value에서 value로 저장된다. 이때 명령어는 저장하지 않고 인자들만 저장한다.

* 1. **Int get\_input()**

변수 설명

🡪int i: input의 길이를 실시간으로 저장하고, MAX\_INPUT\_SIZE를 넘으면 종료하게 한다.

함수 설명

Getchar()를 사용하여 character값을 하나씩 받아온다. MAX\_INPUT\_SIZE를 넘지 않는 선에서 ‘\n’이 입력되기 전까지 input에 저장하며 만약 ‘\n’이 입력되면 이 자리에 ‘\0’을 저장한다.

* 1. **Void FreeMemory()**

변수 설명

🡪node \*temp: history에 저장된 노드들을 하나씩 free해주기 위한 노드 포인터

🡪H\_node \*temp2: hash table에 저장된 노드들을 하나씩 free해주기 위한 노드 포인터

함수 설명

Input과 memory, history와 hash table과 같이 동적으로 할당된 데이터를 프로그램을 종료하기 전에 free해주어야 된다. Hash\_table같은 경우는 0~MAX\_TABLE까지를 돌면서 각 linked list의 값들을 free해준다.

* 1. **Void Help()**

변수 설명

🡪없음

함수 설명

명령어들을 모두 출력해준다. 함수가 끝나기 전에 메모리에 help를 호출했다는 것을 저장한다

* 1. **Void Quit()**

변수 설명

🡪없음

함수 설명

프로그램을 종료하는 함수이다. 종료하기 전에 FreeMemory()를 호출하여 동적할당 된 메모리 값들을 free해준다.

* 1. **Void Dir()**

변수 설명

🡪Dir \*dir: opendir 함수를 이용하여 현재 디렉토리의 스트림을 열고, 그 스트림에 대한 포인터를 저장한다.

🡪Struct dirent \*ent : dirent 구조체는 파일, 디렉토리가 가지고 있는 정보를 담은 구조체이다. Readdir을 이용하여 현재 참조하고 항목의 포인터를 저장한다.

🡪Struct stat temp: stat 구조체는 파일의 상태 및 속성에 대한 정보를 담은 구조체이다. Buf.st\_mode를 이요하여 파일의 속성에 접근한다.

함수 설명

현재 디렉토리에 존재하는 파일과 다른 디렉토리들을 출력한다. 디렉토리인 경우 이름 끝에 ‘/’를 붙이고, 실행파일인 경우 ‘\*’를 붙여서 출력한다.

* 1. **Void History()**

변수 설명

🡪node\* curr: 출력할 history에 관한 linkedlist의 현재 node를 가리킨다.

🡪int i: history의 순서를 출력한다.

함수 설명

Add\_History함수를 먼저 호출한다. History를 호출하면 그 또한 history에 추가하여 출력해야 되기 때문이다. 다음 curr에 history의 첫 번째를 node를 나타내는 his\_head->next를 넣는다. 다음, curr가 NULL이 될 때까지 next를 돌면서 linked\_list의 모든 값들을 출력한다.

* 1. **Void Add\_History()**

변수 설명

🡪Node\* new: history linkedlist에 새로운 node를 추가할 때 사용

🡪Char temp\_input[MAX\_INPUT\_SIZE]: 새로운 history node에 저장할 임시 문자열을 저장한다.

함수 설명

History linked\_list에 새로운 node를 추가하는 함수이다. Temp\_input에 temp\_value의 토큰들을 차례대로 이어 붙인다. 다음 new node->name에 이 값을 넣고, history의 마지막 node를 가리키는 his\_last의 next에 new를 넣는다.

* 1. **Void Dump()**

변수 설명

🡪Long start, end: dump에서 출력할 메모리의 시작과 끝 주소를 저장한다.

🡪Long line : start와 end의 값에 따라서 총 출력해야 될 줄의 개수를 저장한다.

함수 설명

dump함수 뒤에 인자의 개수가 0, 1, 2에 따라서 start와 end를 정한다. 인자가 없는 경우에는 address의 값을 넣고, end 값이 없는 경우에는 start+159를 넣는다. 만약 end가 MAX\_MEMORY\_SIZE보다 크거나 같다면 MAX\_MEMORY\_SIZE-1로 바꾼다. 다음 예외 경우인 end가 start보다 작은 경우에는 오류를 출력하고 종료한다. 오류가 없는 경우 line을 계산하고 형식에 맞게 메모리를 출력한다. 인자의 개수가 0인 경우에는 address를 end로 업데이트하며, Add\_History를 호출한다.

* 1. **Void Edit()**

변수 설명

🡪long address: edit할 메모리의 주소를 저장한다.

🡪long change: edit할 메모리에 새로 들어갈 메모리 값을 저장한다.

함수 설명

edit명령어를 수행한다. 각 value에 저장된 토큰을 16진수로 변환하여 address, change 변수에 저장한다. 다음 주어진 메모리에 주어진 값을 넣는다. Add\_History를 호출한다.

* 1. **Void Fill()**

변수 설명

🡪long start: edit할 메모리의 시작 주소를 저장한다.

🡪long end: edit할 메모리의 끝 주소를 저장한다.

🡪Long change: edit할 메모리들에 새로 들어갈 값들을 저장한다.

함수 설명

Value에 저장된 토큰들을 가지고 start, end, change의 값들을 저장한다. 만약 start의 주소가 end의 주소보다 더 크다면 오류를 출력하고 종료한다. 마약 오류가 발생하지 않았더라면 start부터 end까지의 메모리를 change로 변환한다. 마지막 Add\_History를 호출한다.

* 1. **Void Reset()**

변수 설명

🡪없음

함수 설명

Memory의 값을 모두 0으로 초기화한다. 우선 memory를 free하고 calloc을 사용하여 새롭게 메모리를 할당한다. Add\_History를 호출하고 종료한다.

* 1. **Int Check\_Address()**

변수 설명

🡪int size: check하고자 하는 주소의 길이를 저장한다.

🡪int flag: 주소의 길이가 5보다 큰지 확인하는데 사용된다.

함수 설명

주소 값의 각 자리를 돌면서 각 자리가 16진수에서 사용할 수 있는 수인지 확인한다. 즉, 0~9 또는 a~z 또는 A~Z사이의 값인지 확인한다. 단하나라도 이 범위를 만족하지 않는다면 오류를 출력하고 0을 반환한다. 또한 주소의 길이가 5보다 크다면 이 또한 메모리의 범위를 벗어나기 때문에 오류를 출력하고 0을 반환한다. 유효한 주소 값임이 판단되면 1을 반환한다.

* 1. **Int Check\_Value()**

변수 설명

🡪int size: 메모리 값으로 check할 값의 크기를 저장한다.

🡪int flag: 메모리 값으로 check할 값의 길이가 2보다 큰지 확인하는 변수이다.

함수 설명

메모리 값으로 들어온 값을 각 자리별로 16진수에서 사용할 수 있는 값인지 확인한다. 즉, 0~9, a~z, A~Z사이의 값인지 확인한다. 만약 단 하나라도 그렇지 않은 값이 있다면 오류를 출력하고 0을 반환한다. 또한 길이가 3을 넘는다면 이는 메모리 값으로 들어올 수 없음으로 오류를 출력하고 0을 반환한다. 위의 조건을 다 만족하면 메모리 값으로 만족됨으로 1을 반환한다.

* 1. **Int Get\_Hash()**

변수 설명

🡪int hash: 반환할 값을 저장한다.

🡪int size: hash값을 구하고자 하는 target의 길이를 저장한다.

함수 설명

랜덤하게 만든 hash식을 통해서 target의 hash값을 생성하고 반환한다.

* 1. **Void Add\_to\_hash\_table()**

변수 설명

🡪H\_node\* new: hash\_table에 새로 추가할 H\_node를 생성하기 위한 변수

함수 설명

인자로 받은 opcode, mnemonic, format을 새로 만든 new에 저장한다. 다음 해당 hash값에 해당하는 hash\_table에 new를 추가하는데, 만약 이미 hash\_table[hash]에 저장된 값이 있다면, new의 next에 이미 있는 값을 저장하고 new를 hash\_table[hash]의 첫 H\_node로 한다. 만약 저장된 값이 없다면 그냥 new를 추가한다. 이로 인해서 hash\_table[hash]에 linked\_list형태로 H\_node들이 저장된다.

* 1. **Void Print\_opcodelist()**

변수 설명

🡪H\_node\* temp: 출력할 hash\_table을 가리킬 때 사용한다.

함수 설명

Hash\_table의 0~MAX\_TABLE을 각각의 linked\_list들을 돌면서 해당 H\_node에 해당하는 값들을 출력한다. 마지막에 Add\_History함수를 호출하고 종료한다.

* 1. **Void Find\_opcode()**

변수 설명

🡪int hash: 찾고자 하는 mnemonic에 해당하는 hash값을 저장한다.

🡪H\_node\* temp: hash\_table을 돌면서 찾고자 하는 mnemonic이 있는지 확인하기 위한 H\_node

함수 설명

hash값에 get\_hash를 이용하여 찾고자 하는 mnemonic에 해당하는 hash값을 저장한다. Hash\_table[hash]의 linked\_list를 돌면서 찾고자하는 mnemonic이 있는지 확인한다. 만약 있다면 출력하고 Add\_History함수를 호출하고 종료한다. 만약 없다면 오류를 출력한다.

1. **전역 변수 정의**
   1. **char \*input**

🡪처음 “sicsim>” 뒤에 오는 모든 입력을 저장한다. 이는 뒤에서 temp\_value에 단어별로 토큰을 저장하는데 사용된다.

* 1. **unsigned char\* memory**

🡪메모리 영역을 저장하는 공간. 이후에 동적할당을 이용하여 (16x65536)byte의 가상 메모리 공간을 생성한다.

* 1. **value[3][10]**

🡪temp\_value를 통해서 들어온 입력을 파싱한 이후, 올바른 명령어가 들어왔다고 판단하면 value에는 명령어 뒤의 인자가 들어온다. 단, 들어오기 전에 인자로 들어올 수 있는가를 확인한다. 예를 들어서 주소 값이라면 Check\_Address함수를 통해서 확인 후 만족하는 경우에만 저장된다.

* 1. **value\_num**

🡪value에 들어간 토큰의 개수를 저장한다.

* 1. **temp\_value[4][15]**

🡪input을 작은 단위로 나눠서 저장한다. 공백을 기준으로 나눠서 저장하며, 뒤에 ‘,’가 붙은 입력은 ‘,’를 없애고 저장한다. 이렇게 저장함으로써 유효한 명령어인지부터, 유효한 명령어 뒤의 인자인지까지 확인이 가능하다.

* 1. **temp\_value\_num**

🡪temp\_value에 저장된 토큰의 개수를 저장한다.

* 1. **Address**

🡪dump뒤에 인자가 없는 dump명령어가 호출되었을 때, 출력할 시작 주소를 저장한다. 한번 호출 이후에는 end의 주소를 저장하여 계속 업데이트한다.

* 1. **node 구조체**

🡪node 구조체는 History를 저장하기 위한 linked list 구현에서 사용된다. 안의 변수로는

Char name[MAX\_INPUT\_SIZE]

Struct node\* next

가 있는데, name은 말 그대로 올바른 명령어와 인자를 저장하는 배열이고 next는 다음 node를 가리키는 pointer역할을 한다.

* 1. **H\_node 구조체**

🡪H\_node 구조체는 hash\_table을 만들기 위한 linked list 구현에 사용된다. 안의 변수로는

Char key[MAX\_KEY]: mnemonic값을 저장한다.

Char format[MAX\_FORMAT]: 입력된 format 형식을 저장한다.

Int value: opcode를 저장한다.

Struct H\_node\* next: 다음 H\_node를 가리킨다.

가 있으며, 마지막 next를 통해서 linked list를 만들 수 있다. 입력은 opcode.txt에서 읽어와서 저장한다.

* 1. **Node\* his\_head**

🡪history를 저장하기 위한 linked list의 머리를 저장한다. 이는 History명령어가 입력되었을 때, history를 차례대로 출력할 때 사용된다.

* 1. **Node\* his\_last**

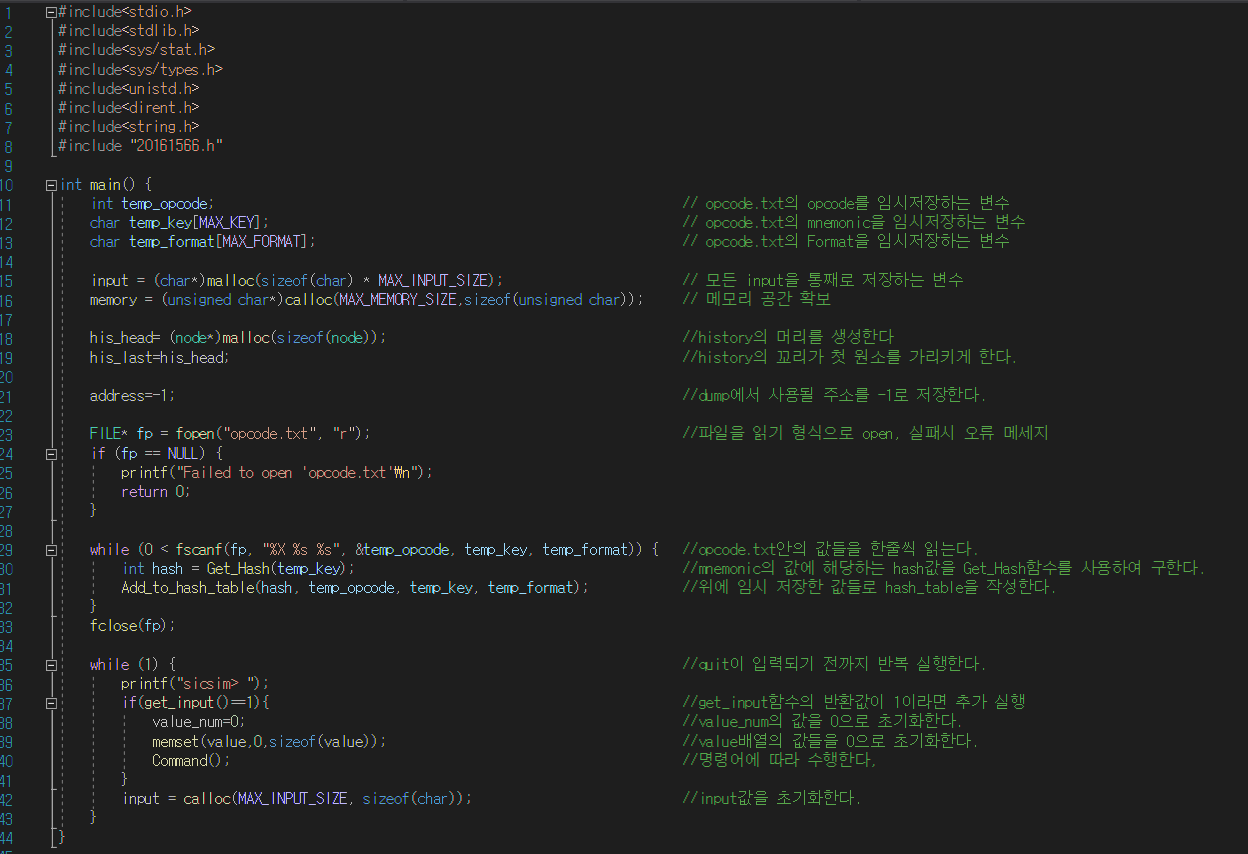
🡪history linked list의 마지막을 저장한다. Add\_History함수가 실행되었을 때, 순서대로 저장돼야 하는 특성 때문에 마지막 node를 알고 있어야 된다. His\_last가 이러한 역할을 수행한다.

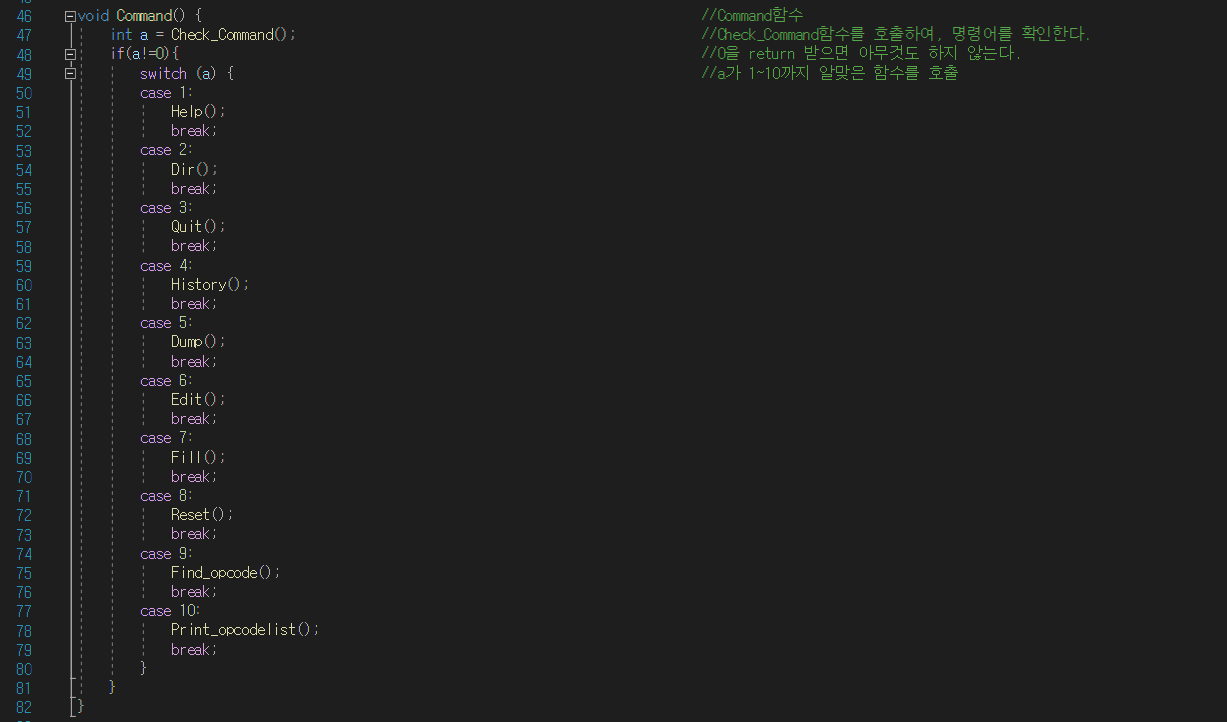
* 1. **H\_node\* hash\_table[MAX\_TABLE]**

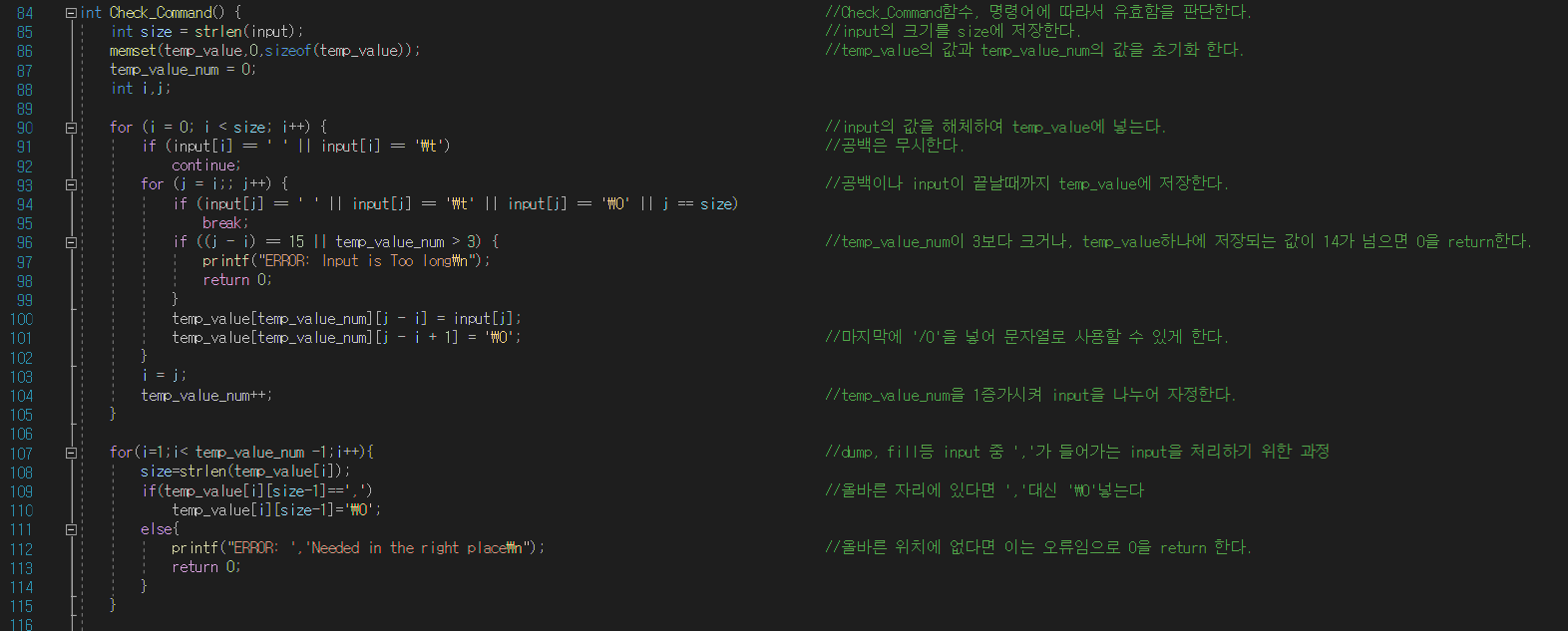
🡪hash\_table을 저장한다. 포인터를 배열로 선언하여 0~MAX\_TABLE(20)만큼 포인터를 선언하고, 만약 hash값이 같은 mnemonic이 들어오면 hash\_table[hash]에 linked list로 이어서 저장하게 된다. 즉, hash\_table[0]은 hash값이 0인 mnemonic들의 head를 가리킨다고 할 수 있다.

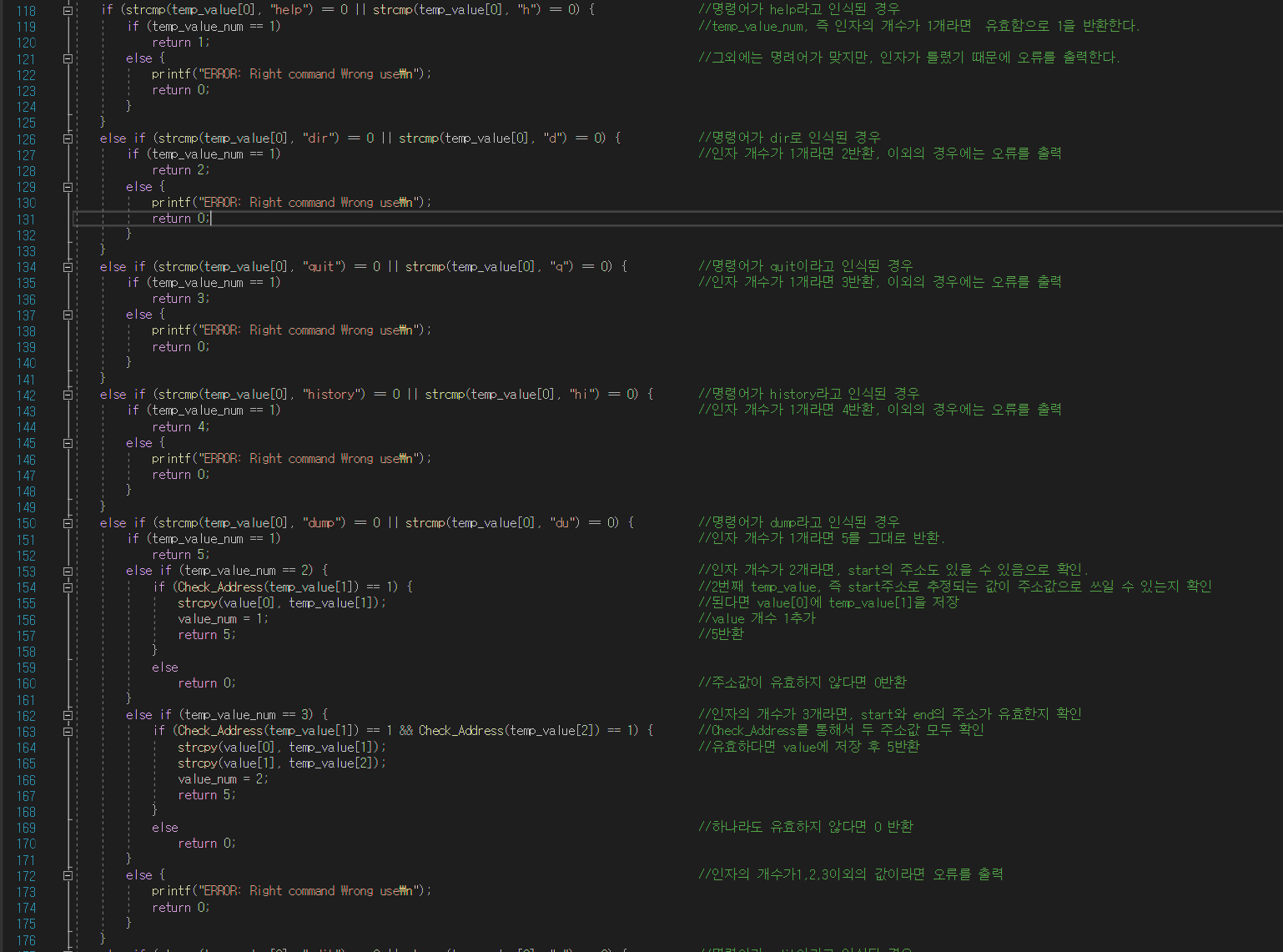
1. **코드 설명**

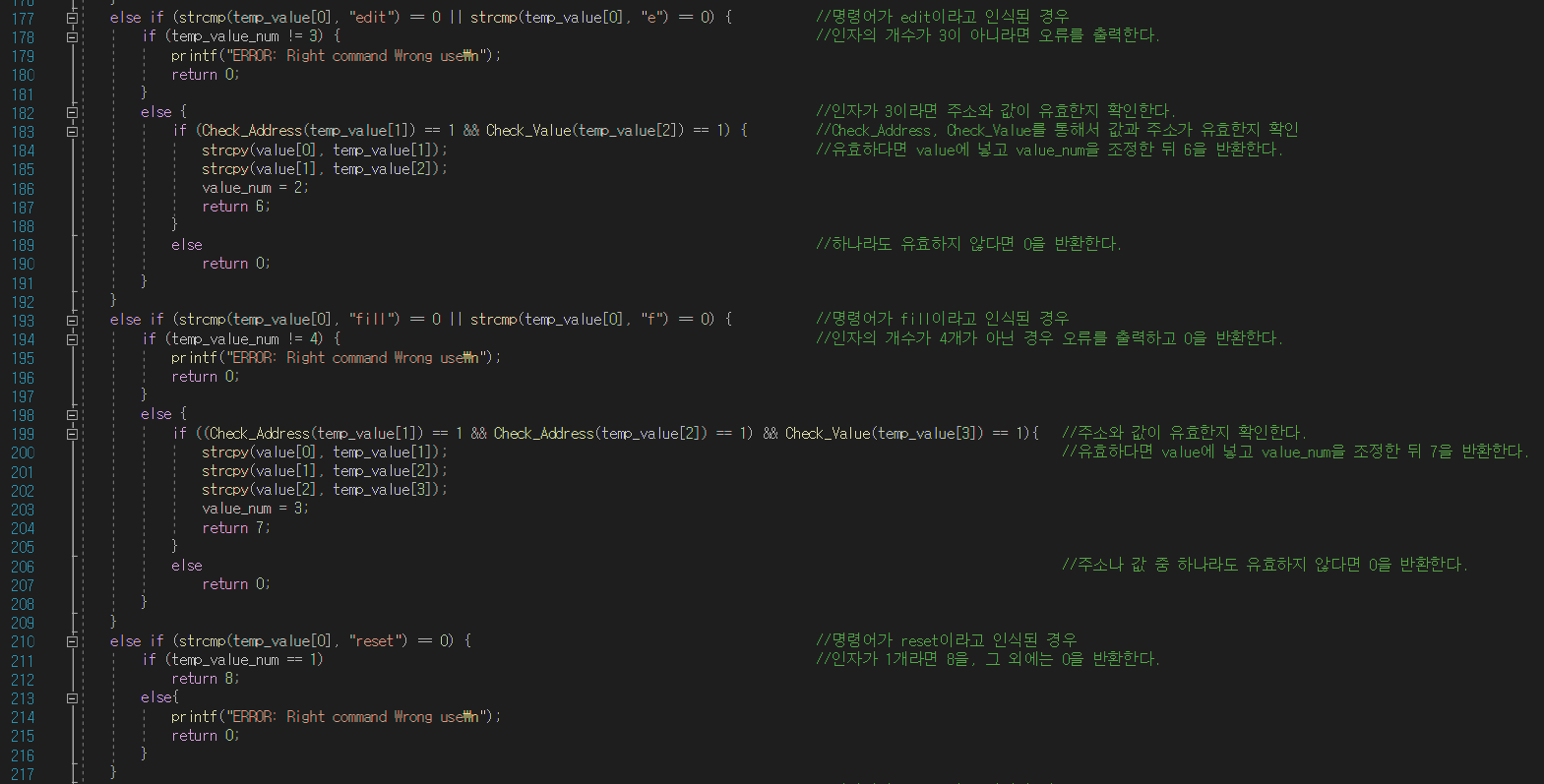
**<20161566.c>**

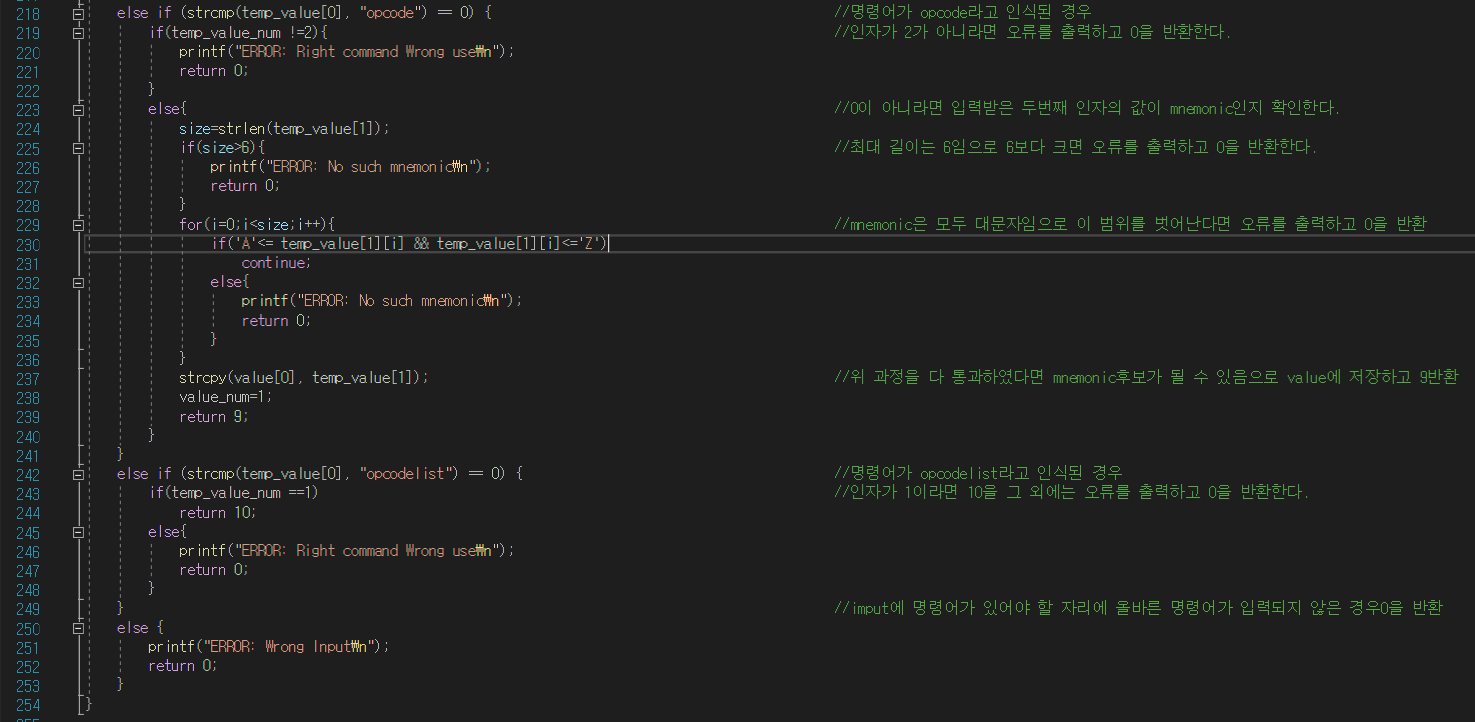


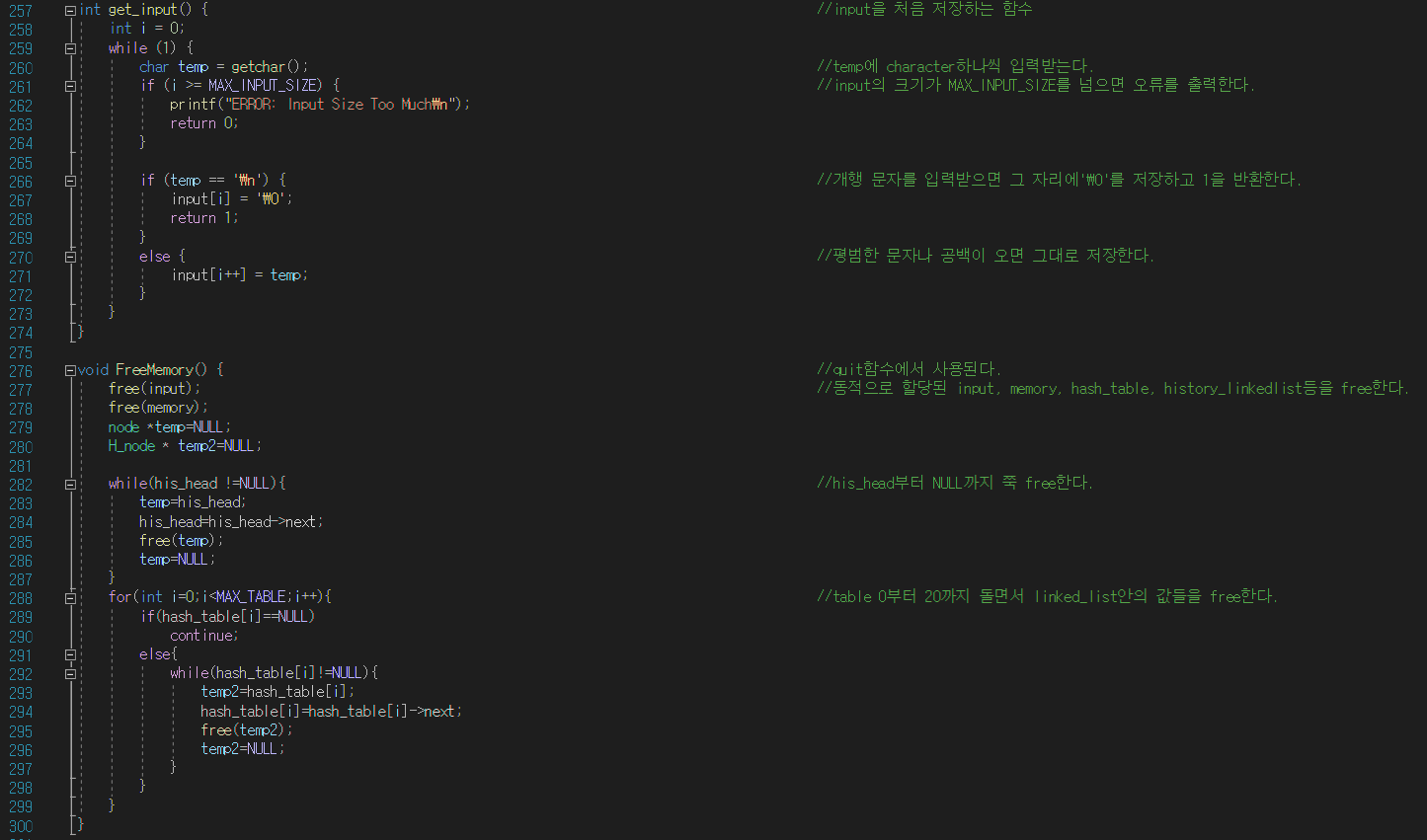


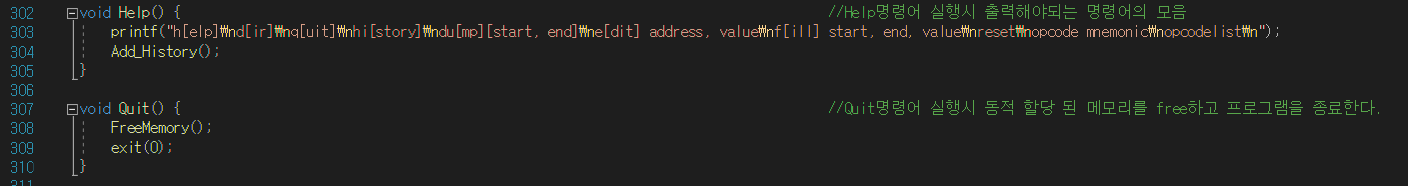


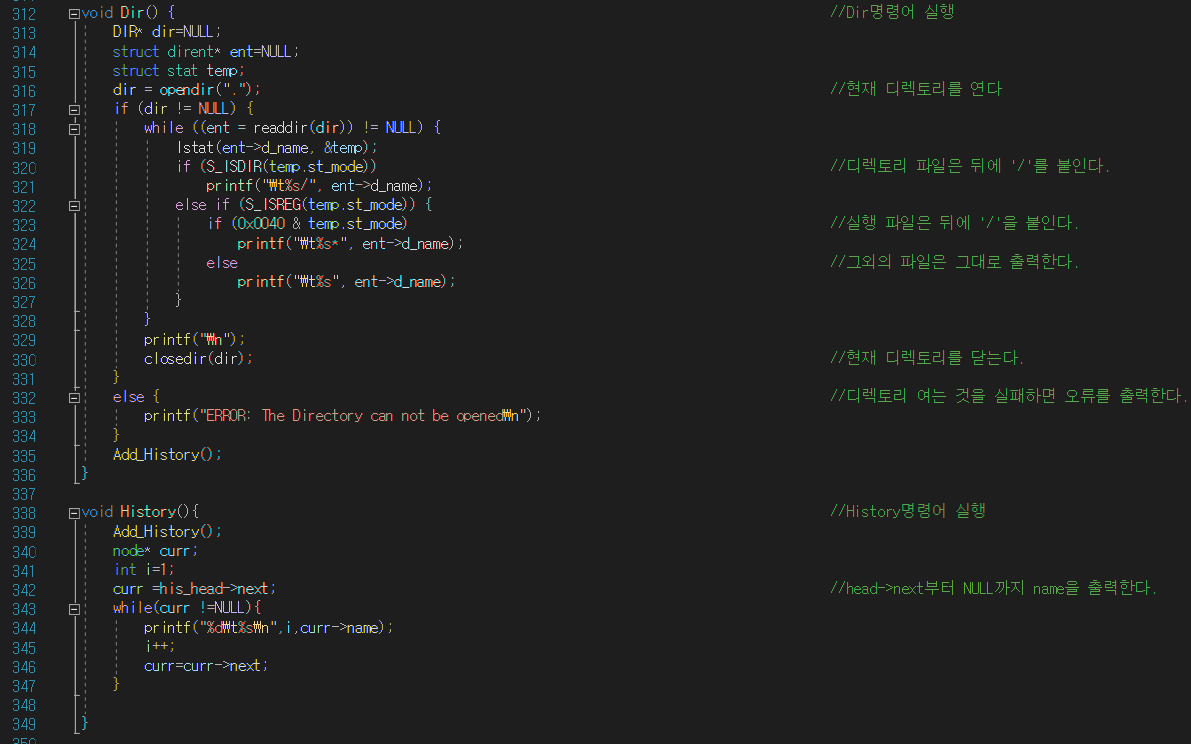




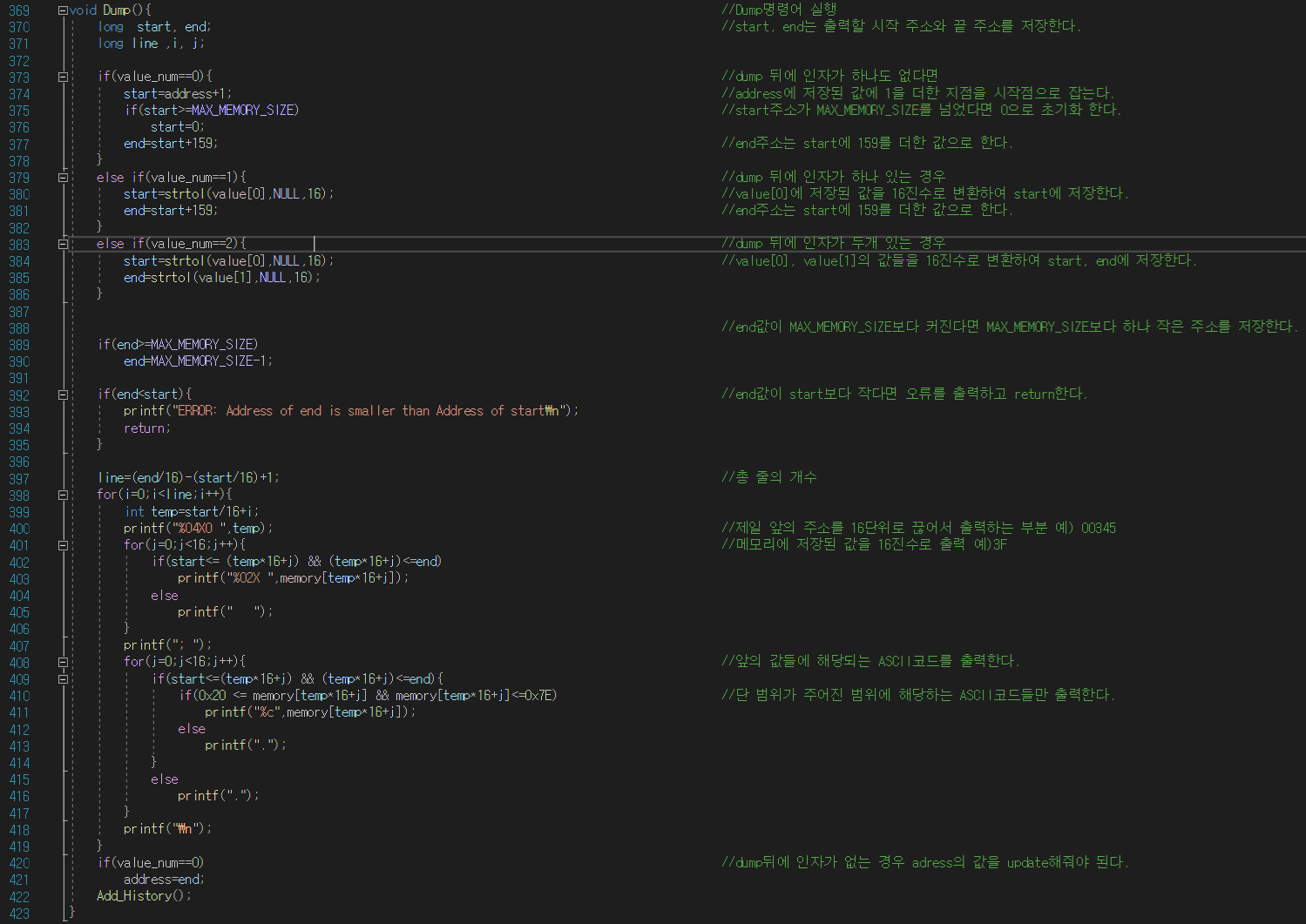


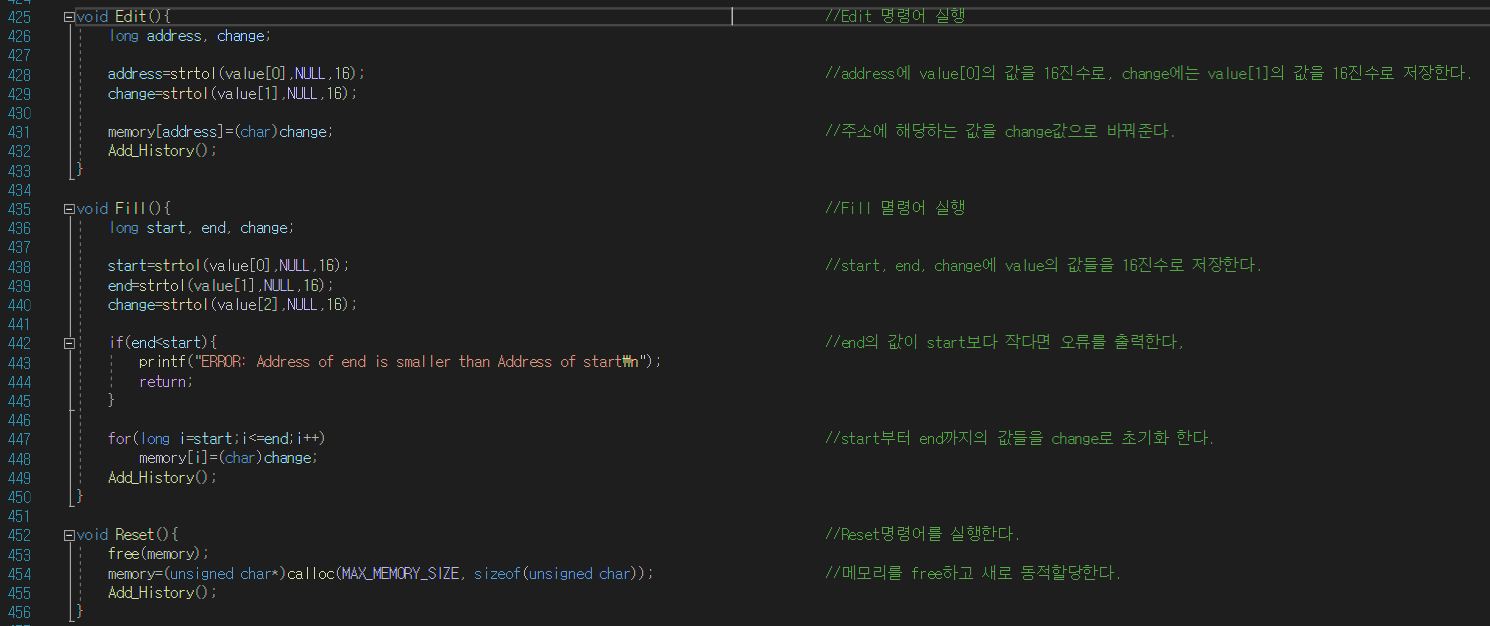


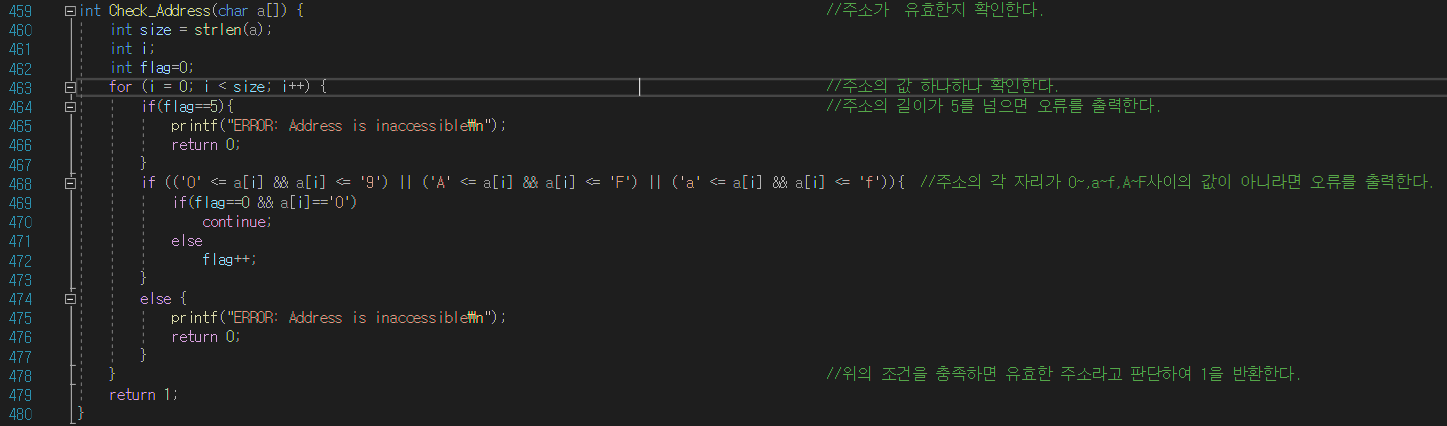


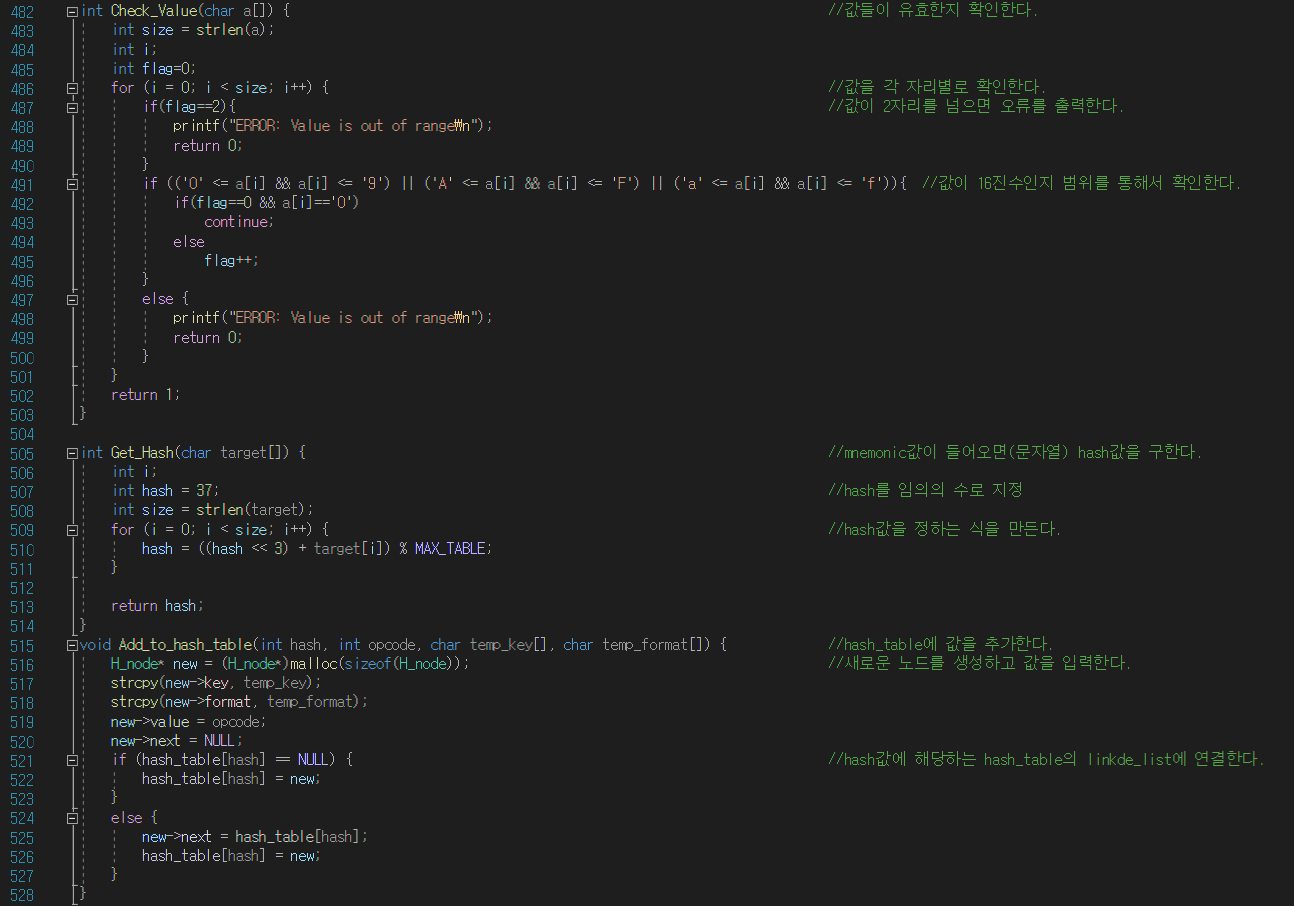


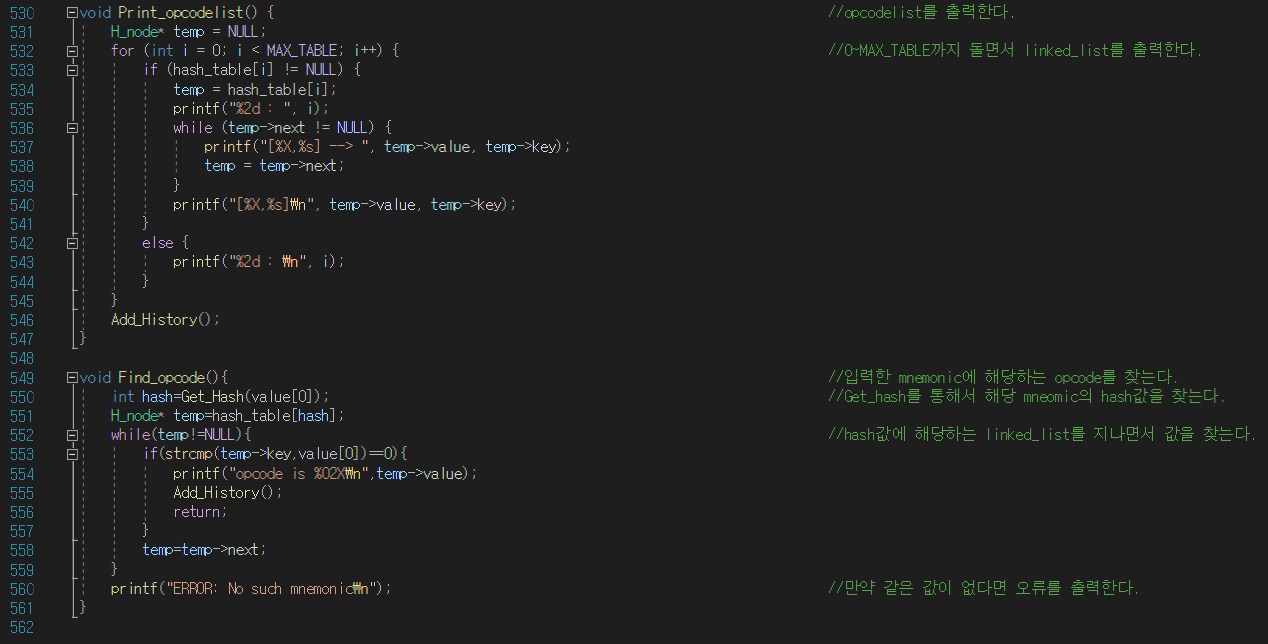




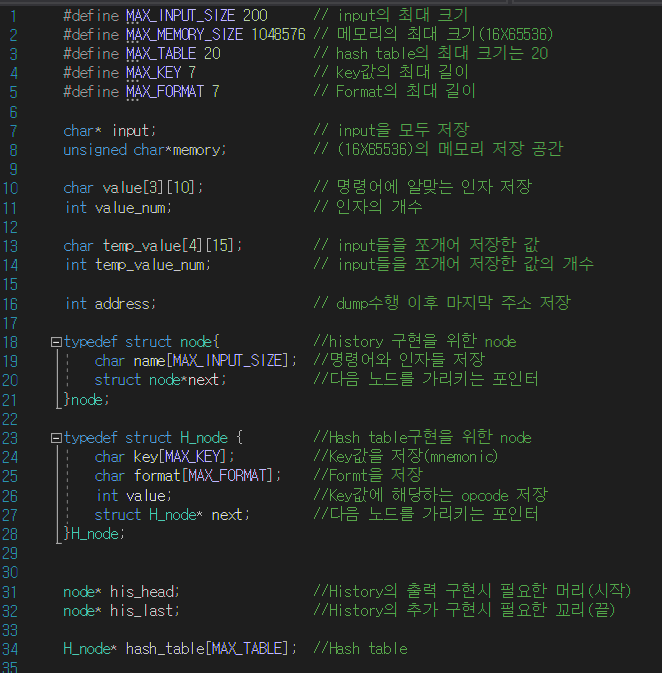


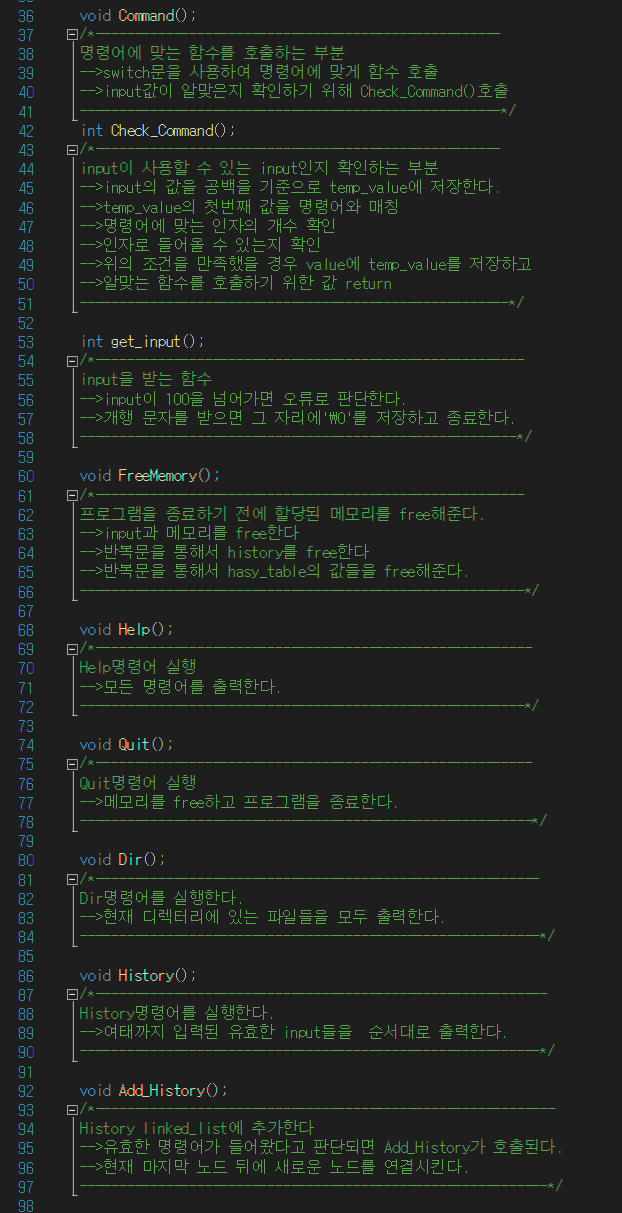




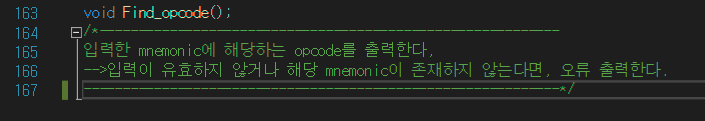


**<20161566.h>**









**<Makefile>**

