C Programming Basic

Minseok Song

Handong Global University

Minseok.H.Song@gmail.com

Index

1 Introduction

C언어의 기본적인 골격과, 그 구조에 대해서 말하는 챕터입니다. C 언어를 사용해 본 적이 있고, 어느정도 코딩을 할 줄 아는 사람을 대상으로 작성하였으니, 참고 바랍니다.

1.1 C 언어의 Basic Struct

```
1  #include <stdio.h>
2
3  int main() {
4      printf("Hello, world!\n");
5      return 0;
6  }
```

1-1_Hello_world.c

C언어를 처음 접하게 되면 만나는 가장 기본적인 코드입니다. 다른 많은 책에서 해당 코드를 접하셨을 것이고, 많은 해석들을 보았을 것입니다. 마찬가지로, 여기서도 C언어의 구성을 보기 위하여 위 코드를 분석해 볼 것입니다. 다만, 각 라인에 있는 코드가 어떤 의미를 뜻하는지 전부 설명할 예정이니, 필요한 부분만 보시는 것을 추천합니다.

`1  #include <stdio.h>`

- **#** 문자는 컴파일 시에 전처리기에게 해당 라인을 처리해 달라는 의미를 가지고 있습니다. (컴파일러와 전처리기에 관한 내용은 3.2 컴파일러 파트를 참고해 주시기 바랍니다.)
- **include** 라는 명령어는 전처리기에게 다른 파일의 내용을 해당 위치에 붙여넣기 하라는 명령어입니다. (관련 내용 또한 3.2 컴파일러 파트를 참고해 주시기 바랍니다.)
- **< … >** 라는 기호는 해당 파일이 C언어 표준 Header 파일이므로, 표준 Header 파일이 들어있는 System이 정의한 폴더에서 파일을 찾으라는 기호입니다.
- **stdio.h** 라는 파일은 Standard Input / Output의 줄인말로써, 표준 입/출력에 관한 내용이 포함되어 있는 Header 파일입니다.

`3  int main() {`

- **int** 타입은 정수를 나타내는 타입입니다. 이는 변수 또는 함수를 선언할 때 사용하는데, 위 코드에서는 함수를 선언하는 용도로 사용되었습니다. 함수를 선언할 때 사용되는 자료형은 해당 함수가 int형 값을 return 한다는 표현입니다. (자세한 내용은 1.3 함수 파트에서 설명합니다.)
- **main** 문자는 함수의 이름을 정의합니다. 선언한 함수의 이름이 main임을 뜻하며,

특히 함수의 이름이 main인 경우는 C언어에서 프로그램이 시작하는 위치를 말합니다. (자세한 내용은 3.1 메모리 파트를 참고해 주시기 바랍니다.)

- **(…)** 기호는 해당 함수가 어떤 인자(parameter)를 받는지에 대해 나타냅니다. 위 예제에서는 아무런 인자도 받지 않는다는 표현입니다.
- **{** 기호는 함수의 내용이 시작됨을 알립니다. 그와 동시에, 새로운 Scope을 설정합니다. (관련 내용은 3.1 메모리 파트를 참고해 주시기 바랍니다.)

```
4   printf(“Hello, world!\n”);
```

- **printf** 문자는 함수의 이름입니다. Console에 문자를 출력할 때 쓰이며, stdout을 통해 출력합니다. stdout이란 Standard output의 약어이며, 입출력을 위한 Stream입니다. 기본적인 Stream으로는 stdin (Standard Input), stdout, stderr (Standard Error)가 있으며, printf 함수는 stdout을 사용합니다.
- **(…)** 기호는 해당 함수를 호출한다는 의미입니다. 함수를 호출하며 동시에 괄호 안에 들어간 인자(parameter)들을 넘겨주는 역할을 합니다. (함수를 호출하는 것을 Function Call이라고 표현합니다.)
- **“Hello, world!\n”** 문자열은 printf를 호출함에 있어 주어지는 인수(argument)입니다. printf는 첫 번째 인자로 문자열을 입력받으므로, “…”를 통해 문자열을 넘겼습니다.
- **;** 기호는 한 라인의 끝을 의미합니다. 컴파일러가 한 문장의 끝을 해석할 수 있도록 하는 구분자(delimiter) 역할을 합니다.

```
5   return 0;
```

- **return** 문자는 예약어로써, 현재 진행중인 함수를 끝내고 본인을 Function Call 한 객체에게 값을 돌려줍니다. 단, main에 한해서 return은 프로그램의 종료를 의미하며, 프로그램이 어떻게 종료되었는지를 나타냅니다.
- **0** 숫자는 return되는 값을 의미하며, main에 한해서 0은 프로그램이 정상적으로 종료되었음을 나타냅니다. 프로그램이 비정상적으로 종료되었으면 0이외의 값을 리턴하는 것이 보편적입니다.

```
6   }
```

- **}** 기호는 함수나 한 Scope의 끝을 의미합니다. 해당 문자를 만나 한 Scope이 끝나게 되면 메모리에 존재하는 Stack에 지역변수들이 모두 삭제됩니다. (관련 내용은 3.1 메모리 파트를 참고해 주시기 바랍니다.)

위에서 나타난 것처럼, C언어는 Header 파트, Function 선언 및 Function 내용으로 이루어져 있습니다. C언어를 이용한 프로그램 프로그래밍에서는 이 틀을 벗어나지 않으며, 커널 프로그래밍이나 임베디드 프로그래밍 등 특별한 분야에 한해서 이 틀을 벗어난 코딩을 하게 됩니다.

1.2 변수 (Variable)

변수는 프로그래밍에서 가장 중요한 역할을 맡고 있습니다. 변수가 존재함으로써 무언가를 저장할 수 있는 기능이 동작하고, 저장 및 불러오기를 통해 다양한 일들을 수행할 수 있게 되었습니다. C언어에서는 변수를 선언하고 저장하며 Function call의 인자로 넘겨주는 등 다양하게 사용됩니다. 이번 파트에서는 변수를 어떻게 사용하는지에 대해 배웁니다. 변수가 어떻게 저장되고 운용되는지는 3.1 메모리 파트에서 확인하시기 바랍니다.

```
<Type> <Var name> [ = <Value> ] ;
```

변수의 선언은 위와 같은 형태로 이루어집니다. 먼저 변수의 Type을 설정하고, 변수의 이름을 지정한 뒤 세미콜론을 입력합니다. 이 때, 값을 할당할 수도, 할당하지 않을 수도 있습니다. 선언과 동시에 값을 할당하는 것을 초기화(Initialization)라고 하며, 이를 하지 않을 시에는 쓰레기 값(Garbage value)이 들어가 있습니다.

변수의 Type은 여러 종류가 있으며, 한글로는 '자료형'이라고 해석합니다. 각 자료형 별로 표현할 수 있는 값의 범위와 종류가 다르므로, 저장하려는 값의 특성에 맞게 자료형을 정하시면 됩니다. 대표적으로 int, long, char, float, double 등이 있으며, 각각 정수, 큰 정수, 문자, 실수, (조금 더 정교한)실수를 나타냅니다.

변수의 이름은 몇 가지 조건을 제외하고 선언할 수 있습니다. 그 조건은 **숫자로 시작**할 수 없으며, **알파벳과 숫자와 언더바(_)**로만 구성될 수 있고, **예약어**를 사용할 수 없습니다. 예약어란 return이나 int 등과 같이 C언어에서 미리 지정해 놓은 키워드를 의미합니다. 아래 예제에서 변수의 선언에 관한 내용을 볼 수 있습니다.

```c
1  int main() {
2      int var1 = 1;
3      long var2;
4      char var3 = 'c';
5      double var4;
6  }
```

1-2_Variable.c

2 `int var1 = 1;`
- var1 변수는 정수형 데이터를 저장하고, 그 값으로는 1을 지정해 주었습니다.

3 `long var2;`
- var2 변수는 큰 정수형 데이터를 저장합니다. 이 때 값을 지정해주지 않았으므로, var2에는 쓰레기 값이 저장되어 있습니다.

4 `char var3 = 'c';`
- var3 변수는 문자 하나를 저장합니다. 문자를 표현하기 위해 '…' 기호를 사용하였습니다.

```
5   double var4;
```

- var4 변수는 정교한 실수를 저장합니다. 초기화를 하지 않았기에 쓰레기 값이 들어가 있습니다.

위 예제에서 두 개의 변수는 초기화가 되었고, 두 개의 변수는 초기화가 되지 않았습니다. 초기화가 되지 않은 변수는 어떤 값이 들어있을지 몰라 사용하기에는 큰 위험이 따릅니다. 당장 사용하지 않을 예정이더라도, 0과 같은 값으로 초기화를 해 주는 습관을 들이는 것이 좋습니다. 개인적으로, **초기화(Initialization)는 매우 중요**하다고 생각합니다.

1.3 함수 (Function)

```
<Type> <Function name> ( <parameters > ) {
        <Function body>
}
```

함수의 선언은 위와 같이 이루어집니다. 먼저 함수가 return 할 값을 Type으로 선언합니다. 다음, 함수의 이름을 지정하고, 소괄호 내부에 어떤 값들을 받을 것인지 인자(Parameter)를 기입합니다. 이후, 중괄호를 열어 함수의 행동을 정의합니다. 함수가 return 한다는 것은 함수를 부른 대상에게 함수가 일을 처리하고, 처리 결과를 돌려준다는 것을 의미합니다. C언어에서는 처리 결과가 숫자 혹은 문자로 표현될 수 있기에, int나 float, char 등의 변수 타입으로 함수를 선언합니다.

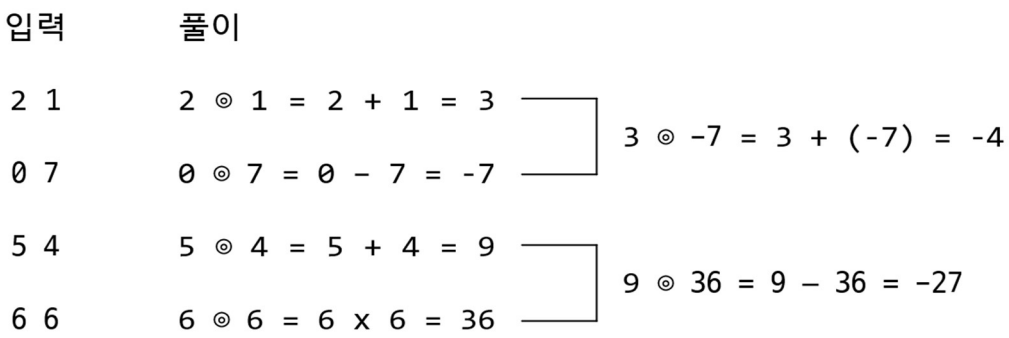
함수란 특정한 역할을 지정하여 그 역할에 몰두할 수 있도록 코드를 세분화하는 역할을 합니다. 마치 공장에서 생산품을 만들 때 각 구역별로 하는 일이 지정되어 있고, 세분화되어 있는 것처럼 프로그램에서도 각 구역별로 하는 일을 지정하고, 세분화하는 것입니다. 다만, 처음에 프로그래밍을 하게 될 때는 굳이 함수를 사용하지 않아도 잘 되는데 함수를 써야하는 이유를 모르겠고 그냥 쓰라고 해서 쓰는 경우가 많이 있습니다. 함수를 사용하는 이유는 크게 두가지입니다. **코드를 재활용하는 것**과 **코드의 가독성(Readability)** 때문입니다.

코드를 재활용한다는 뜻이 처음에는 어떤 뜻인지 받아들이기 힘들 것입니다. 아래 문제와 풀이, 그리고 함수를 활용한 풀이를 보여드릴 것입니다. 한번 보시고, 함수를 어떻게 활용하였는지 확인하시면 좋겠습니다.

- 문제. 수학에서 새로운 연산자를 정의했다. A ⊙ B는 A가 크면 A + B, B가 크면 A – B, 같으면 A x B라고 한다. 처음 몇 개의 A, B 쌍이 입력되는지에 해당하는 N을 입력받고, 이후 N번만큼 A, B가 입력된다. 이 때, (2n 번째 A ⊙ B) ⊙ (2n+1 번째 A ⊙ B)의 결과를 출력하시오. (단, N은 짝수이다.)
- 입출력 예제

입력	출력
4 2 1 0 7 5 4 6 6	-4 -27

- 설명



- 풀이 (함수 없이)

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main() {
5       int N = 0, A = 0, B = 0, result = 0;
6       int *arr = NULL;
7       int i = 0;
8
9       scanf(“%d”, &N);
10      arr = (int *)malloc(sizeof(int) * N);
11
12      for (i = 0; i < N; i++) {
13          scanf(“%d %d”, &A, &B);
14          if (A > B)
15              result = A + B;
16          else if (A < B)
17              result = A – B;
18          else
19              result = A * B;
20          arr[i] = result;
21      }
22
23      for (i = 0; i < N; i += 2) {
```

```
24          if (arr[i] > arr[i + 1])
25              result = arr[i] + arr[i + 1];
26          else if (arr[i] < arr[i + 1])
27              result = arr[i] – arr[i + 1];
28          else
29              result = arr[i] * arr[i + 1];
30
31          printf(“%d\n”, result);
32      }
33
34      free(arr);
35      return 0;
36  }
```

1-3_problem_without_function.c

```
10  arr = (int *)malloc(sizeof(int) * N);
```

- 위 코드는 int arr[N]; 선언과 동일합니다. 자세한 내용은 동적 메모리 내용을 참고해 주세요.

```
23  for (i = 0; i < N; i += 2) {
```

- i를 0부터 N까지 2씩 증가시켜가며 아래 내용을 반복하라는 의미입니다.

해당 코드는 특별한 함수 없이, 모든 코드를 main 안에 적은 경우입니다. 특히나, 14 ~ 20번 라인과 24 ~ 31번 라인은 코드 내용은 비슷한데 두 번 중복되어 있습니다. 이러한 경우를, 하나의 함수로 통일해서 작성할 수 있습니다.

- 풀이 (함수 포함)

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int new_op(int A, int B) {
5       int result = 0;
6       if (A > B)
7           result = A + B;
8       else if (A < B)
9           result = A – B;
10      else
11          result = A * B;
12      return result;
13  }
14
15  int main() {
16      int N = 0, A = 0, B = 0, result = 0;
17      int *arr = NULL;
```

```c
    int i = 0;

    scanf("%d", &N);
    arr = (int *)malloc(sizeof(int) * N);

    for (i = 0; i < N; i++) {
        scanf("%d %d", &A, &B);
        arr[i] = new_op(A, B);
    }

    for (i = 0; i < N; i += 2) {
        result = new_op(arr[i], arr[i + 1]);
        printf("%d\n", result);
    }

    free(arr);
    return 0;
}
```

1-3_problem_with_function.c

이전에 보았던 중복되는 내용을 하나의 함수로 합침으로써 코드도 간결해지고, 무엇보다 기존에 A와 B, arr[i]와 arr[i + 1]로 사용했던 내용을 인수로 넘김으로써 같은 코드를 재활용했다고 할 수 있습니다.

또한, 가독성 부분에서도 이점이 있습니다. 1-3_problem_without_function.c 파일에서는 14 ~ 20번 라인, 24 ~ 31번 라인이 무슨 역할을 하는 코드인지 모르지만, 1-3_problem_with_function.c 파일에서는 해당 코드가 new_op (New Operation)이라는 함수명을 통해 새로운 연산이라는 것을 확인할 수 있습니다.

"무슨 역할을 하는 코드인지 모른다" 라는 말이 이상하게 보이실 수도 있습니다. '당연히 문제에서 새로운 연산을 정의했고, 그 연산을 구현하였으니 누구나 새로운 연산을 표현했다는 것을 알고 있지 않은가?'라고 생각하시기 때문이라고 예상해봅니다. 이 문제에 한해서는 그 생각이 틀리지는 않습니다. 다만, 앞으로 프로그래밍을 하다 보면 혼자서 코드를 작성하는 것보다는 여러 사람이 같이 코드를 짜게 되는 일이 더 많습니다. 이 때, 함수로 내용을 구분해 놓지 않는다면 다른 사람이 코드를 볼 때 코드를 해석하기 위해 더 많은 시간을 쏟아야 하고, 더 많은 노력을 쏟아야 합니다. 어디가 잘못되었는지, 어디를 고쳐야 하는지 알기 어렵습니다. 만약 기능별로 함수로 모두 정리되어 있다면, 잘못된 내용이 들어있는 함수만 코드를 해석해도 되고, 다른 내용은 살펴보지 않아도 됩니다. 마치 백과사전에 가 ~ 하 까지 내용이 차곡차곡 정리되어 있는 것처럼, 함수로 기능을 나누는 것은 어질러져 있는 코드들을 잘 정리하는 것이라고 말씀드리고 싶습니다. 특히 남들과 공유하는 코드를 작성할 때는 **코드를 기능별로 함수를 만들어 정리**하는 것이 매우 중요하다고 생각합니다.

2 Statements

2.1 조건문 (If statement)

프로그래밍이라는 소프트웨어 분야가 발전하기 전에는 하드웨어가 발전하고 있었습니다. 하드웨어의 발전은 많은 문제들을 해결할 수 있었지만 치명적인 문제가 하나 있었는데, 바로 사용자의 입력에 따라 다른 행동을 하게 하는 것이 너무 어려웠다는 것입니다. 이 때문에 유동적으로 사용자의 입력에 반응하기 위한 소프트웨어가 필요했고, 자연스럽게 프로그래밍이라는 분야가 발전하기 시작하였습니다.

조건문은 사용자의 입력이나 특정 내용을 통해 반응하고, 어떻게 대처할 것인지 분기를 나누는 것을 말합니다. (분기를 나눈다는 말은 한자 표현입니다. 흐름을 제어한다, 선택지를 가른다 정도로 해석할 수 있습니다.) C언어에서는 if 라는 명령어를 통해 조건문을 작성할 수 있으며, else if나 else와 함께 사용됩니다.

```
if (condition) {
    True statement
}
else if (condition) {
    True statement
}
else {
    False statement
}
```

if 옆에 condition이 참(True)인 경우에는 첫 번째 True statement를 실행하고, 아닌 경우에는 두 번째 else if에 있는 condition을 확인하고, 참인 경우에는 두 번째 True statement를 실행하고, 아닌 경우에는 else에 있는 False statement를 실행합니다. 이 때 else if와 else는 생략이 가능합니다.

2.2 반복문 (Loop statement)

반복문은 while과 for 두 방법이 있습니다. 많은 분들이 언제 while을 사용하고, 언제 for를 사용하는지에 대해 질문을 하시지만, 이 질문은 정답이 없습니다. 본인이 while이 더 사용하기에 편하다면 while을 사용하고, for를 사용하기가 더 편하다면 for를 사용하시면 됩니다.

```
for (initial statement; test statement; update statement) {
    Statements
}
```

for문은 위와 같이 사용할 수 있습니다. 한 가지 주의해야 할 점은 for문 내부에서 실행되는 순서입니다. for문은 내부적으로 initial statement, (test statement, Statements, update statement) 순으로 실행되기 때문에 초기화를 진행하고, 테스트를 한 다음 테스

트가 참인 경우에 for문 내부로 들어갑니다. 그리고, 업데이트를 한 다음 테스트를 하므로 for문에서 한번이라도 내부로 들어가 코드를 실행했다면 마지막 update가 진행된 채로 빠져나옵니다.

사람들이 잘 모르는 내용 중에서, for문의 initial statement와 update statement는 ‘,’ 기호를 통해 여러 구문을 묶어서 사용할 수 있습니다.

```
1   for (i = 0, j = 0; i < 10 && j < 5; i += 2, j++) { … }
```

위와 같이, initial statement에서 i와 j를 함께 0으로 초기화하고, update statement에서 i는 2씩, j는 1씩 증가시키는 것이 가능합니다. 다만, test statement에서는 ‘,’ 기호로 묶는 것이 불가능하고, &&나 ||같은 연산자로 연결해 주어야 합니다.

```
while (condition) {
    Statement
}
```

while문은 위와 같이 하나의 condition만으로 반복을 제어하는 구문입니다. 주의하실 내용은 condition이 참인 경우에 아래 Statement를 실행하므로, 항상 조건을 잘 생각해야 합니다. (이 부분에서 많은 사람들이 실수를 합니다.)

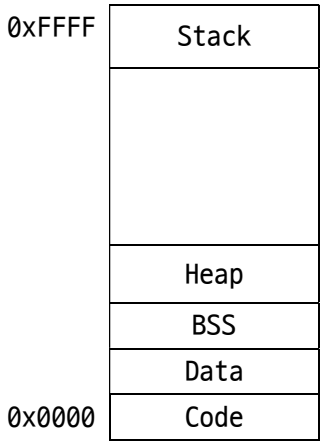
3 Mechanism

이 챕터의 내용은 이해하기 어려울 수 있습니다. 주로 Operating System(운영체제)에서 배우는 내용이고, 심화 내용의 공부를 원하시는 분들에게 어떤 내용을 공부할 수 있는지에 대한 가능성을 알리기 위한 파트입니다. 내용 또한 방대하기 때문에, 자세히 설명할 수 없어 간략하게 설명합니다. 따라서, 한번 훑어 읽어보시는 정도로만 넘어가시는 것을 추천드립니다.

3.1 메모리 (Memory)

C언어는 High level language이기 때문에 메모리를 어떻게 사용하는지에 대해 주의 깊게 생각하지 않고 코드를 작성해도 괜찮습니다. 다만, 메모리가 어떻게 운용되는지에 대한 개념을 알게 되면 조금 더 효율적인 코드 작성과 디버깅에 대한 이해도가 올라갈 수 있기 때문에 메모리 운용에 관한 내용을 들어 보시기를 추천합니다.

프로그램이 실행되면 메모리(RAM)에 프로그램이 사용할 수 있는 공간이 할당됩니다. 이 때 공간이 무작정 할당되는 것은 아니고, 특정한 형태를 가지고 할당됩니다. 그 형태는 우측에 있는 대로 할당되는데, Code 영역은 실제 실행되는 코드가 Binary 형태로 들어가 있는 부분이고, Data 영역은 초기화된 전역변수 및 static 변수들, BSS에는 초기화 되지 않은 전역변수 및 static 변수들, Heap은 동적 할당된 메모리가 존재하는 곳이고, Stack은 함수 호출이나 지역변수 선언 등에 사용되는 메모리입니다.



Stack은 메모리를 사용하면 사용할수록 아래 방향으로 공간이 확장됩니다. 처음 시작 시에 main 함수가 스택 영역의 제일 위쪽에 할당되고 (정확하게는 틀린 말이지만, 이해를 돕기 위해 이렇게 작성하였습니다.) main 함수에서 function call을 하게 되면 현재 변수를 모두 저장한 이후 새로운 function의 scope을 Stack에 확장합니다. 이후 함수가 끝나게 되면 function의 scope을 모두 삭제하고, 이전에 저장해 두었던 변수를 모두 가져옵니다. 그렇게 Stack은 공간이 늘었다 줄었다를 반복하며 프로그램이 실행되고, 프로그램이 종료되는 순간 Stack의 공간은 0이 됩니다.

함수를 벗어나도 메모리 공간이 유지될 수 있도록 따로 메모리 공간을 저장하는 곳이 Heap입니다. Heap은 malloc을 통해 메모리 공간을 위쪽 방향으로 확장하는데, 따로 free를 해주기 전까지 메모리 공간이 할당되어 있습니다. (여기서 말하는 Heap은 자료구조의 Heap과는 다른, 그저 명칭이 Heap인 공간입니다.)

Stack은 아래 방향으로 자라고, Heap은 위쪽 방향으로 자라게 됩니다. 간혹 malloc을 했을 때 NULL이 반환되는 경우가 있는데, Stack이 너무 많이 자라거나 Heap이 너무 많이 자라 공간이 부족한 경우 NULL이 반환됩니다. 반대로, Recursion을 잘못 설정하여 무한으로 Recursion에 빠지게 되는 경우 Stack이 계속 자라 Heap 영역을 침범하게 되는데, 이 때 발생하는 오류가 Segmentation fault입니다.

3.2 컴파일러 (Compiler)

흔히 말하는 컴파일러란 C언어로 작성된 파일을 읽고, 실행시킬 수 있는 파일로 만들어 주는 프로그램을 말합니다. 다만, 이는 정확한 표현이 아닙니다. C언어 코드로 작성된 파일을 읽고 실행 가능한 파일로 만드는 이 과정을 통틀어서 “빌드(Build) 한다”고 표현하는데, 이 빌드 과정에는 여러 과정이 합쳐져 있습니다.

- 전처리기 (Preprocessor) / 전처리 과정 (Preprocessing)
- 컴파일러 (Compiler) / 컴파일 과정 (Compile)
- 어셈블러 (Assembler) / 어셈블 과정 (Assemble)

- 링커 (Linker) / 링킹 과정 (Linking)

총 네 과정을 거쳐 .c 파일이 .exe 혹은 .out 파일로 나오게 됩니다. 이 파트는 각각의 과정이 어떤 역할을 맡고, 어떤 코드에 영향을 미치는지에 관한 내용입니다.

전처리기는 .c 파일에서 #으로 시작하는 부분을 담당합니다. #include 혹은 #define 등이 해당 내용에 포함됩니다. #include는 다른 파일의 내용을 붙여넣기 하는 역할을 하고, #define은 전처리 과정에서 사용되는 변수를 정의하거나 코드상의 내용을 치환하는 역할을 합니다. 또한, #ifdef 혹은 #ifndef 등과 함께 사용되는 #else, #endif 등은 전처리 과정에서 사용되는 변수가 정의가 되었는지 여부에 따라 빌드하는 과정에서 코드를 삭제하기도 합니다. 이를 매크로라고 하며, 매크로로 인라인 함수를 정의하기도 하는 등 다양한 코딩의 편의성을 제공합니다.

컴파일러는 이전에 전처리기를 통해 주석이나 #이 붙은 내용이 모두 삭제되고, 순수한 C언어 코드만 남아있는 상태의 코드를 어셈블리 언어로 치환해주는 역할을 합니다. C언어는 High level language라서 바로 기계어로 만들 수 없기 때문에, Low level language인 어셈블리 언어로 치환하는 과정을 거치게 됩니다.

어셈블러는 컴파일러를 통해 변환된 어셈블리 언어를 기계어로 변환해 주는 역할을 합니다. 이 때, 생성된 기계어로 된 Binary 파일을 Object 파일이라고 하고, 더욱 자세히는 Relocatable Object File이라고 하는데, 이는 ELF(Executable and Linkable Format) 폼으로 구성되어 있습니다. 어셈블러를 통해 Binary 파일이 되면 바로 실행할 수 있을 수도 있지만, 여러 파일들을 묶어서 컴파일을 하는 경우 다른 Objcet 파일에 정의된 함수를 사용하는 경우도 있고, 다른 Binary 파일에 정의되어 있는 경우도 있기 때문에 해당 함수나 변수의 주소값은 기계어로 번역되지 않고, ASCII 코드의 형태로 남아있습니다.

링커는 위에서 생성된 Relocatable Object File들을 모두 묶어서 남아있는 ASCII 코드들을 제거하고, 하나의 Executable File로 완성해 주는 역할을 합니다. 자세한 내용을 다루기에는 내용이 어렵기 때문에, 더욱 자세한 내용을 보고 싶으신 분들은 인터넷을 통해 배우시기를 추천드립니다.