※ 변수 생성 시점은 값이 변수에 들어갈 때이다. 실제 값이 대입될 때, 공간이 잡히므로 그 때 기계가 변수 생성을 한다.
※ 메모리는 1차원이다. 직렬화 되어 있기 때문이다. 따라서, 2, 3차원이 아니다.

***데이터**

- 데이터의 타입 : int, short, char, float, double, long double

int: 4 byte(32 비트 = 2^32 개를 나타낼 수 있음, 대략 42 억 9 천만개)
short: 2 byte(16 비트 = 2^16(65536)개에 해당)
char: 1 byte(8 비트 = 2^8(256)개에 해당)
float: 4 byte
double: 8 byte(2^64)
long double: 12 or 16 byte

데이터 타입에 **unsigned** 가 붙으면 음수가 존재하지 않는다. unsigned 가 없으면 음수값이 존재하게 된다. 이 unsigned가 중요한 이유는 컴퓨터의 레지스터(register)는 여러 개의 gate들이 모여서 구성되는데, 자료형보다 큰 수 혹은 작은 수를 사용할 경우 문제가 생기기 때문이다.
오류가 생기는 상황을 보자. Unsigned가 붙어서 음수가 존재하는 상황일 때, 0은 양수에 포함 된다. 그래서 256(char 타입은 -128~127 이다.)개의 수를 담을 수 있다면, 양수는 `0~127'이고 음수는 '-1~ -128'이 된다(항상 음수의 범위가 1개 더 크다.)
이 때, 128을 입력하게 된다면, 어떻게 될까? > -128이 된다. 컴퓨터가 앞으로 더 나아가고 싶지만 범위를 초과하여 갈 수 없기 때문에, 남은 한 칸이 범위 뒤로 가서 뒤의 가장 작은 수가 되는 것이다. 이는 **오버플로우(overflow)** 현상인데, 정해진 범위를 초과할 경우 발생한다. **언더플로우(underflow)**는 정해진 범위 미만일 경우 발생한다.

[**오버플로우(overflow)**]

어떤 데이터 타입이 표현할 수 있는 최대값이 있고, 이 범위를 벗어날 경우 오히려 맨 아래로 내려가는 현상.

[**언더플로우(underflow)**]

위와 마찬가지인데 표현할 수 있는 최소값에서 더 아래로 내려갈 경우 반대로 맨 위로 올라가는 현상.

char 타입은 -128 ~ 127 이므로 이 범위를 기준해서 보자면 127 + 1 = -128 / -128 - 1 = 127이 되는 현상이 대표적이다. 더 보자면, 127 + 2 = -127 / -128 - 2 는 126 이 된다.
또한, 2진수에서 우리가 흔히 알고 있는 숫자 1과 -1을 더하면 0이 되는 것에 문제가 있다. 2진수에서는 1과 -1을 더한다고 0이 되지 않는다. 예시를 한 번 보자. (32비트로 계산하게 되면 조금 복잡하므로 8비트로 예시를 들어보자.) 먼저, 127을 살펴보자.

127= 64 + 32 + 16 + 8 + 4 + 2 + 1

0 1 1 1 1 1 1 1 > 맨 앞은 부트비트 자리이다. 0은 양수이고 1이면 음수이다.

1 1 1 1 1 1 1 1 > -127이므로 부호비트 자리가 1이다.

1 / 0 0 0 0 0 0 0 0 > 8비트이므로 앞의 1은 없는 것과 같다. 이렇듯, 음수와 양수를 더하면 0이 되어야 하나, 다른 수는 그렇지 않다.

문제가 없어 보이나, 1의 경우를 살펴보자.

0000 0001 +1
1000 0001 우리의 잘못된 -1
1000 0010 우리의 잘못된 0 >위의 127처럼 0이 되지 않는다. 왜일까?

이렇듯, 처음에 범 할 수 있는 오류가 맨 앞의 부호 비트만 바꾸면 -1 이 되겠지 하는 오류다. 이런 문제를 해결하기 위해 2의 보수라는 것이 나온다.

2의 보수[two's complement]

2진수(binary number)의 보수. 보수는 원래 컴퓨터에서의 연산에 「음(負)」의 수를 표현하기 위하여 만들어지는 것이다. 2진수 n자리(n비트)에 대하여 2^n을 기수로 하는 경우의 보수. 실용적으로는 1의 보수(2진 기수법에 있어서 보수를 취할 때 각 자리의 반대수 (0→1, 1→0)를 취하는 보수.)에 1을 더하여 얻게 된다. 양의 정수에 2의 보수를 취해야 그것이 음의 정수 값이 된다.

이와 같은 방법 외에도 빠르게 음수만 획득하는 방법이 있다. 0으로 만드는 것이 목표일 경우, 맨 처음 나오는 1을 놔두고 (방향은 뒤에서 앞이다.) 반전시키는 방법이다.

0000 0001 +1
1111 1111 -1 > 1을 놔두고 반전시킨다.
10000 0000 0 > 맨 앞의 1 은 버린다. 8비트이므로..

확인을 해보기 위해서 5 와 10, 28 등에 실험해보자

0000 0101	+5	0000 1010	+10	0001 1100	+28	
1111 1011	-5	1111 0110	-10	1110 0100	-28	
1 0000 0000	0	1 0000 0000	0	1 0000 0000	0	> 맨 앞의 1은 버린다.

*Character Literal 문자 리터럴

컴퓨터 프로그램의 소스 코드 내에서 단일 문자의 값을 표현하기 위한 프로그래밍의 리터럴 유형을 말한다. 문자들 자체에 의해 값이 주어지는 문자열이다. 예를 들면, 숫자 리터럴 7은 7의 값을 가지며, 문자 리터럴 'CHARACTERS'는 CHARACTERS의 값을 갖는다. 아스키(ASCII) 코드표에 따른 문자에 대응하는 수이다. 기본적으로 'A'는 65이며 'a'는 97이다. 문자 양쪽에 '을 붙여 표현한다. 문자 간의 덧셈, 뺄셈도 가능하다. 컴파일러가 사용한다. 이 아스키 코드는 왜 사용될까? 문자가 숫자로 치환이 되기 때문이다. 이 것이 가능하기 때문에 암호화에 매우 효율적이다. 암호화는 왜 필요한가? 기본적으로 우리가 일상에서 사용하는 모든 정보는 유선이 아니라 무선을 통해서 많이 보급되고 있다. 전화, Wi-Fi, LTE 등이 그 예이다. 문제는 SDR 이라는 근래 아주 핫한 무선 분야다. 이것을 활용하면 무선상의 모든 데이터를 가로챌 수 있다. 암호화가 되어 있지 않다면 id, pw가 모두 유출될 수 있다. 그렇기 때문에 암호화가 필요하다.

전용 문자 데이터 유형을 갖는 언어에는 일반적으로 문자 리터럴이 포함된다. 여기에는 C, C ++, Java 및 Visual Basic 이 포함되지만 Python 또는 PHP는 포함되지 않는다. 문자 데이터 유형이 없는 언어는 일반적으로 길이가 1 인 문자열을 사용하여 문자 데이터 유형이 수행하는 것과 동일한 목적을 제공한다. 이는 언어의 구현 및 기본 사용을 단순화하지만 프로그래밍 오류에 대한 새로운 범위도 포함한다.

문자 리터럴을 나타내는 일반적인 규칙은 문자열 리터럴에 큰 따옴표 "를 사용하는 것과는 대조적으로 문자 리터럴에 작은 따옴표 '를 사용하는 것이다 (예 :'a '는 한 문자를 나타내는 반면 " a " 는 길이가 1 인 문자열 a를 나타낸다.) 즉, '는 문자 1만을 나타내고 "는 주소(배열)를 나타낸다.

컴퓨터 메모리, 저장 장치 및 데이터 전송 내의 문자 표현은 특정 문자 인코딩 체계에 따라 다르다. 예를 들어, ASCII 체계는 컴퓨터 메모리의 단일 바이트를 사용하지만 UTF-8 체계는 인코딩되는 특정 문자에 따라 하나 이상의 바이트를 사용한다.

문자 값을 인코딩하는 다른 방법으로는 ASCII 코드 값 또는 유니 코드 코드 포인트와 같은 코드 포인트에 대한 정수 값을 지정하는 방법이 있다. 정수 리터럴을 문자로 변환하거나 이스케이프 시퀀스를 통해 직접 수행 할 수 있다.

* 문자는 %c, 문자열은 %s, 포인터는 %p

문자를 입력할 때는 작은따옴표(' ')안에 작성해야 한다. 문자열을 입력할 때는 큰따옴표(" ") 안에 작성한다. %s는 소스 안에 있는 그대로 문자열을 나타낼 때 사용한다. 포인터는 주소값을 구하는 서식문자이다.

* 가장 최초로 암호화를 사용한 사람은 로마의 시저 장군이다. 군사 기밀 문서를 중간에 유출될 것을 우려하여 암호화 시키는데 모든 문자에 + 3 을 했다. 그래서 붙은 암호화 별명이 시저 암호화인 것이다. 받은 문서를 다시 - 3 해서 복호화하면 실제로 적은 내용을 확인할 수 있었다. 세계 1 차 대전, 2 차 대전 등에서도 이 기법은 계속해서 발전해왔고 지금도 진행 중이다. 재밌었는 실사례라면 DES 가 하는 알고리즘이 있는데, 이 암호화가 누구도 풀 수 없다고 사람들이 자랑을 하고 다녔다. 그런데 인도의 어떤 사람이 앉아서 명상을 하다가 이 알고리즘의 해를 찾아서 풀어버린다. 해를 찾았다는 것이 y = x + 2 와 같이 x 에 값을 넣으면 바로 y[암호]가 튀어나온다는 뜻이다. 현업으로는 통신 장비 암호화 프로그램이 있다.

아스키 코드

초창기에 다양한 방법으로 문자를 표현했는데 호환 등 여러 문제가 발생했다. 이를 해결하기 위해 1962년 안시(ANSI)가 정의한 미국 표준 정보 교환 코드이다. 현재 이 코드가 일반적으로 사용되고 있다. 이 코드는 7비트의 이진수 조합으로 만들어져 총 128개의 부호를 표현한다. 아스키 코드의 처음 32개(0-31)는 프린터나 전송 제어용으로 사용되고 나머지는 숫자와 로마글자 및 도량형 기호와 문장 기호를 나타낸다.

아스키(ASCII)는 7자리의 2진 코드인데 1비트의 패리티 비트를 추가하여 8개의 비트로 많은 컴퓨터에 사용되고 있다. 아스키(ASCII)코드의 비트 번호는 오른쪽에서 왼쪽으로 부여한다. 아스키(ASCII)는 여러 가지로 다양하게 사용되는 코드에 따른 정보 호환성의 제한이나 불편을 덜고, 컴퓨터 대 컴퓨터, 시스템의 통신을 단순화하고 표준화하기 위해 통신 장비의 사용자들과 자료 처리 사업자들이 협력하여 만들었다.

C 연산자란?

연산자는 메모리 혹은 레지스터 내의 데이터들을 가공하기 위해 C/C++에서 제공하는 기본적인 키워드들이다. =, +, -, *, /, %와 같은 것들이 있다. ++, --와 같은 증감 연산자와 <, >, ==, !=, <=, >=와 같은 관계 연산자와 &&, ||, !와 같은 논리 연산자가 존재한다. 형변환, 그리고 sizeof와 같은 연산자와 그 외의 복합 연산자와 shift 연산자가 존재한다.

* printf 안에서 '%' 를 출력하고 싶다.

메타 문자 방식(%%)을 사용해서 찍는 방법과 아스키 코드를 이용하는 방법이 있다. 디버깅 용도로 많이 쓰기 때문에 아무거나 상관없다. 아스키코드 '%' 는 숫자로 37 에 해당한다. 그러므로 % 를 출력하고자 한다면 %c 에 37 을 넣어주면 '%' 를 출력할 수 있다(%c 는 char 로 문자 1 개를 출력할 때 사용함).

연산자	의미	연산자	의미	전위, 후위 연산	
+	덧셈	+=	a+=b > a=a+b	연산자	의미
-	뺄셈	-=	a-=b > a=a-b	++a	선 증가, 후 연산
*	곱셈	*=	a*=b > a=a*b	--a	선 감소, 후 연산
/	나눗셈	/=	a/=b > a=a/b	a++	선 연산. 후 증가
%	나누고 나머지는 반환	%=	a%=b > a=a%b	a--	선 연산, 후 감소

* 전위, 후위 연산 : ++ 이 뒤에 오면 더하기 연산이 다음 라인에서 실행되고,
++ 이 앞에 오면 먼저 더하기 연산이 실행된다.

* '=' 연산자를 볼 때 주의할 점 : C 언어 코드 분석 시에는 항상 오른쪽에서 왼쪽으로 보도록 한다. 그리고 '=' 연산자는 결국 어셈블리어의 mov 와 동일하다. 예를 들어, num1 = num2 = num3; 는 num3 의 값을 num2, num1 에 셋팅하는 것과 동일한 역할이다.

연산자	의미	연산자	의미
==	같다. a==b > a와 b는 같다. 양쪽 값이 같을 때 참으로 반환한다.	<<	왼쪽으로 몇만큼 값을 이동시킨다. 이진수이므로 값이 2^가 된다. a<<b=a*2^b
!=	같지 않다. a!=b > a와 b는 같지 않다. 양쪽 값이 다를 때 참으로 반환된다.	>>	오른쪽으로 몇만큼 값을 이동시킨다. 이진수이므로 값이 1/2^이 된다. a<<b=a*1/2^b

* 관계 연산자 : <, >, <=, >=, ==, != 와 같은 연산자는 결과가 참 혹은 거짓이다.

* '>>', '<<' shift 연산

* 비트 연산자 : AND, OR, XOR, NOT, 쉬프트 연산이 존재한다. AND 는 서로 참으로 같을 때만 참(1) 이 된다. OR 는 둘 중 하나만 참이면 참이 된다. XOR 는 서로 달라야만 참이 된다. NOT 은 그냥 뒤집는다.

ex) ~0 = 1

~1000 = 0111

쉬프트 연산은 비트를 이동시키는 연산이다. 결국 2 의 승수로 곱하거나 나누는 연산이다.

10 << 1　　1010 << 1　　1010 << 3

10100 = 20　　1010000 = 80

10 >> 2　　1010 >> 2 = 10 결과적으로 10 진수 2

* 주의 사항 : << 쉬프트는 그냥 곱하면 되어서 문제 없으나, **>> 쉬프트는 나눌 때 소수점 자리를 그냥 버려버린다**.

AND 는 서로 참으로 같으면 참이다. 연산 기호는 '**&**' 이다.

10 & 3

1010

0011 AND

0010 > 2

OR 은 하나만 참이면 참이다. 연산 기호는 ' **|** '이다,

10 | 16

01010

10000 OR

11010 > 26

XOR 은 서로 다르면 참이다. 연산 기호는 '^' 이다.

10 ^ 5　　10 ^ 3

1010　　1010

0101 XOR　　0011 XOR

1111 > 15　　1001 > 9

※ 10000 - 1 = 16 - 1 = 15

※ 이렇듯 같으면 0 다르면 1로 표현되어서 암호화할 때, 항상 XOR 을 사용한다.

NOT 은 참이면 거짓, 거짓이면 참이다. 즉, 반전 시키는 것이다. 연산 기호는 '**~**' 이다.

~10 --> ~1010

000000000....1010 NOT

111111111....0101　　-X = -11　　> 앞에 얼마나 더 있는지 모르기 때문에 반전이 필요하다.

00000000000001011　　11 = X

더 빠르게 하는 방법 : 컴퓨터 시스템은 형식상 양수에 0 을 포함시키므로, 음수가 숫자가 1 더 클 수 밖에 없다. 그러므로 10 을 반전 시키면 -11 이고 100 을 반전시키면 -101 이고 10001237 을 반전시키면 -10001238 이 된다.

usigned char t = 8;

~t = 255 - 8 = 247

* 논리 연산자 : &&, ||, ! 이 존재한다. 특별히 주의할 것은 없다. 관계 연산자와 마찬가지로 결과는 참과 거짓이다. ! 사용시 주의할 것이라면 값이 있으면 거짓, 없으면 참이라는 것만 주의하면 된다. &&은 양쪽 피연산자가 모두 참이면 참(1)으로 반환한다(AND). ||은 양쪽 피연산자 중 하나

이상 참이면 참으로 반환한다(OR). !는 참이면 거짓, 거짓이면 참으로 반환한다. 즉, 반대로 반환한다(NOT).

* 숏컷(shortcut)의 이점 : 기본적으로 if 문을 사용하면 mov, cmp, jmp 형식의 3 개의 어셈블리 코드가 만들어진다. shortcut 을 사용하면 비교하는 cmp 가 사라진다. 특히 ARM 으로 구현할 때 더더욱 이득을 볼 수 있다. 코드 최적화 용도로 사용하는 기법이다.

[코드 만들 시, 통 메인 절대 금지]
1. 가독성이 심각하게 떨어진다.
2. 디버깅, 유지 보수가 매우 심각하게 어렵다.

*sizeof()
sizeof(a)는 a에 대한 자료형의 byte 크기를 나타낸다.

* scanf ()
키보드로 무언가를 입력하고자 한다면 사용한다. printf 와 쌍을 이룬다. C언어에서 Printf가 출력의 대표격이라면, scanf는 대표적인 입력함수라 할 수 있다. **scanf()는 정수 형태로 data를 입력 받고, printf()는 정수 형태로 data를 출력** 한다. scanf() 역시 printf()와 마찬가지로 여러 개의 data를 받을 수 있다. Scanf("%d", &num1)을 보면 정수에 대한 입력이 들어오면, 들어온 입력을 변수 num1에 저장한다.
scanf 의 첫 번째 입력은 "%d" 혹은 "%f", "%lf" 등이 올 수 있다. 그리고 두 번째 입력은 반드시 결과를 받을 주소값을 적어야 한다. 그렇기 때문에 주소값을 의미하는 & 가 들어간 것이다. 여기서 &이 차후 배울 Pointer에서 매우 중요하다. 변수 앞에 &를 붙이면 현재 변수가 Memory상에 위치한 주소값을 얻는다.

```
#include <stdio.h>
  int main(void)
 {
       int num1, num2;
       double real1, real2;

      printf("2개의 정수를 입력하세요 : ");
      scanf("%d%d", &num1, &num2);
      printf("입력된 정수 : %d, %d₩n", num1, num2);

      printf("실수를 입력하세요 : ");
      scanf("%f      %f", &real1, real2);
      printf("입력된 실수 : %f, %f₩n", real1, real2);
      return 0; }
```

와 같이 사용되는 것이 그 예이다. (수는 지정하지 않은 예시이다.)

* if문

if 문은 조건을 지정하고 싶을 때 사용한다. 이 안에는 관계 연산자, 조건 연산자 등등이 올 수 있다. 복잡하게 생각하지 말고 어떤 조건을 만족하는지 안 하는지를 생각하면 된다. 그리고 if(조건)에서 조건이 만족되지 않을 경우에는 else 쪽으로 이동하게 된다. 만약 여기서도 추가 조건을 비교하고자 한다면 else if(또 다른 조건)의 형식으로 여러 조건을 만들 수 있다. 단. 참과 거짓의 지옥은 만들지 않도록 주의한다. 최대 3번까지만 사용하도록 한다. 많이 쓰는 것은 좋지 않은 습관이다.

```
#include <stdio.h>
 int main(void)
 {      int num;

      printf("정수를 입력하세요 : ");
       scanf("%d", &num);

      if(num < 0)
       {          printf("입력된 수는 0 미만₩n");      }

      if(num >= 0)
       {          printf("입력된 수는 0 이상₩n");      }

       return 0;   }
```

수는 정하지 않은 예시이다. if문은 이런 식으로 쓰인다. else if의 경우는 if의 조건이 아닌 다른 조건을 넣고 싶을 경우에 사용한다.

```
#include <stdio.h>
int main(void)
 {      int num;

      printf("정수를 입력하세요 : ");
       scanf("%d", &num);

      if(num == 0)
            printf("입력된 수는 %d₩n", num);
       else if(num > 0)
            printf("입력된 수는 양수 %d₩n", num);
       else
            printf("입력된 수는 음수 %d₩n", num);

      return 0; }
```

선택지 안에 또 다른 선택지를 넣어 코드를 구성하고 싶을 때, 사용하는 것이 **nested if** 문이다.

If 안에 또 다른 if가 있는 경우이다. nested는 집합을 의미하는데, 집합 중에서도 그냥 집합이 아니라 앞의 것을 포함한다는 의미를 가지고 있다. 다중 if는 하나의 조건을 가지고 원하는 문장을 산출하지만 nested if는 조건 안에 또 다른 조건이 있다는 소리이다. 이렇게 중첩으로 if문이 사용된 경우를 nested if 라 한다. 즉, 다중 if는 한 번의 계산으로 원하는 조건의 문장을 찾아 산출되지만, nested if는 조건을 계산하고 또 그 안의 조건을 계산하여 나타내고자 하는 문장을 산출하는 것이다.

*switch문

if 문으로 참과 거짓의 지옥을 만들게 되면, 가독성이 심각하게 떨어진다. 반면, switch는 case 부분에 조건이 있어서 case에 해당하는 조건만 보고 해당 부분만 확인하면되므로, If 문을 이용한 참과 거짓의 지옥보다는 가독성이 올라간다. 즉, 참과 거짓이 많을 경우 쓰는 것이다. 그래서 컴파일러를 switch 문을 써서 만든다. 추가적으로 case 안에서의 동작은 break를 만나기 전까지는 계속된다. 즉, switch 문 안에서

case 1:

printf("~₩n");

break;

로 break;를 적지 않으면 끝나지 않는다는 것이다. **case와 break는 세트**로 붙는다고 생각하면 된다. default문 다음의 break 여부는 중요하지 않다. Default는 조건이 모든 case에 상응하지 않으면 실행되기 때문이다. 그래서 break가 없더라도 switch 문이 끝난다.

* while 문

if 나 switch 와 같이 조건을 활용한다. 단, 작업을 반복시킬 수 있고 조건이 거짓이 될 때까지 반복하게 된다. 다른 추가기능이나 요구 사항 없이 조건만 만족하면 내부 코드를 반복 수행하게 할 수 있다. 즉, 조건이 만족하는 동안 반복하게 된다. 경계(범위)를 주의해야 한다. while문은 조건식으로만 구성되기에 조건을 변화시키는 구문이 반복문 내부 코드나 호출되는 함수에 구현되어 있어야 한다. 터미널 창에서 무한루프가 계속 될 경우 'ctrl + c'로 빠져나오면 된다.

1. 스키장에서 스키 장비를 임대하는 37500원이 든다. 또 3일 이상 이용할 경우 20%를 할인 해 준다. 일주일간 이용할 경우 임대 요금은 얼마일까? (연산 과정은 모두 함수로 돌린다)

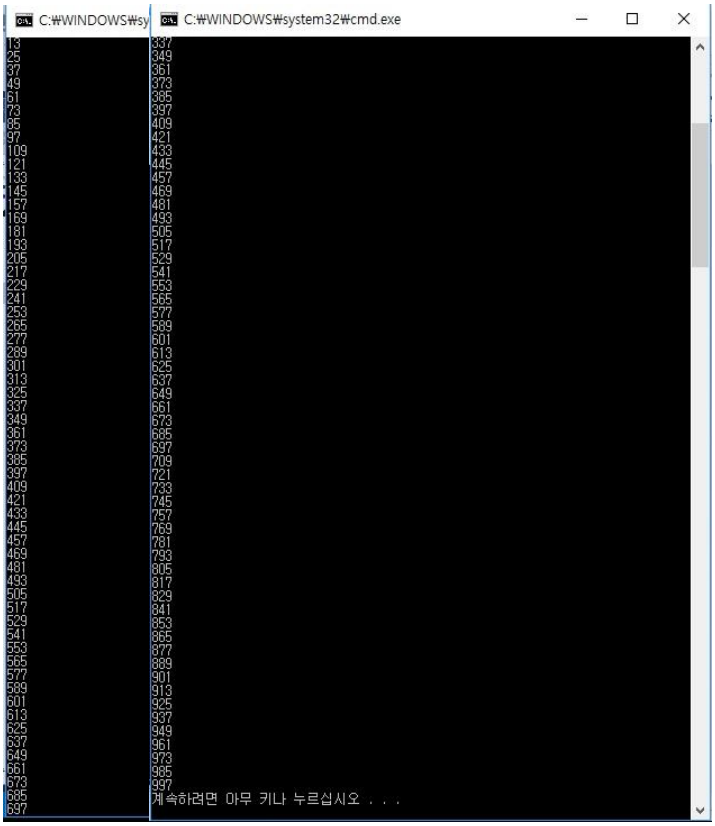
```
#include <stdio.h>
int mul(int num1, int b)
{
    return(num1*b);
}

int main(void)
{
    int num1;
    printf("정수를 입력하세요 :");
    scanf("%d", &num1);

    int b = 37500;

        if (num1< 3)
            printf("%d * %d = %d₩n",num1,b,mul(num1,b));
    else
        printf("%d * %d =%d₩n",num1,b,0.8*mul(num1,b));

        return 0;
}
```

3. 1 ~ 1000 사이에 2의 배수의 합을 구하시오.

```
#include <stdio.h>

void main(void)
{
        int i;

while (i = 0; i <= 10000; i++)
{
        if (i / 2 == 0)
        printf("%d", i)
        i++;
}
```

4. 1 ~ 1000 사이에 4나 6으로 나눠도 나머지가 1인 수의 합을 출력하라.

```
#include<stdio.h>

 int main(void)
 {
     int a = 1;
          while (a <= 1000)
          {
           if (a == 1)
            {
               a++;
            }
          else
            {
               if (a % 4 == 1)
               if (a % 6 == 1)
               printf("%d₩n", a);
               a++;
             }
          }
}
```



5. 7의 배수로 이루어진 값들이 나열되어 있다고 가정한다. 함수의 인자(input)로 항의 개수를 받아서 마지막 항의 값을 구하는 프로그램을 작성하라.

> 죄송합니다.. 이해를 잘 못하겠습니다 ㅠㅠㅠ

7. c로 함수를 만들 때, Stack이란 구조가 생성된다. 이 구조가 어떻게 동작하는지 Assembly Language를 해석하며 기술해보시오. esp, ebp, eip 등의 Register에 어떤 값이 어떻게 들어가는지 등등 메모리에 어떤 값들이 들어가는지 등을 자세히 기술하시오.

```
int mult2(int num)
{
          return num * 2;
}

int main(void)
{
          int i, sum = 0, result;
          for( i = 0; i <5; i++)
```

```
                sum += i;
        result = mult2(sum);
        return 0;
}
```

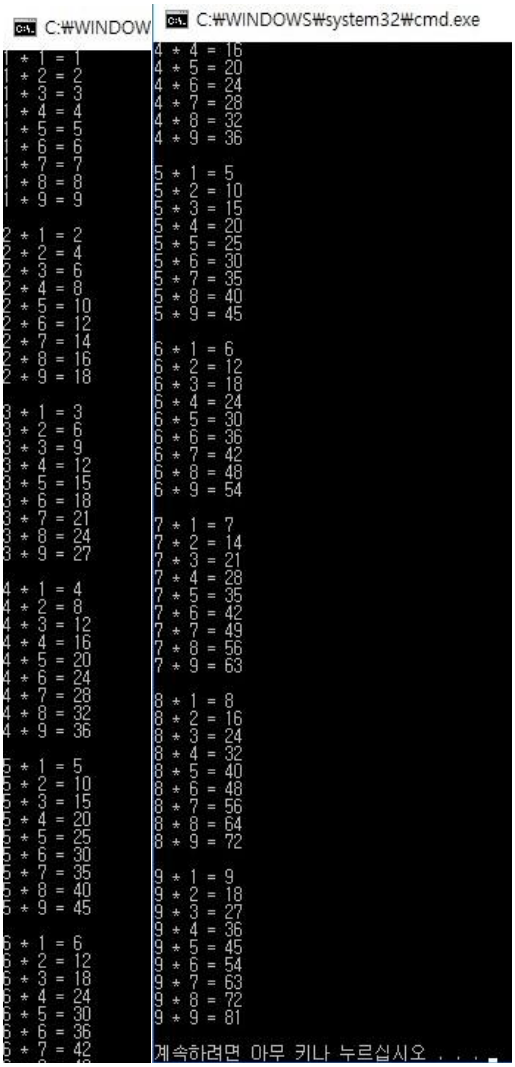
```
   Dump of assembler code for function main:
=> 0x00000000004004e4 <+0>:     push   %rbp
   0x00000000004004e5 <+1>:     mov    %rsp,%rbp
   0x00000000004004e8 <+4>:     sub    $0x10,%rsp
   0x00000000004004ec <+8>:     movl   $0x0,-0x8(%rbp)
   0x00000000004004f3 <+15>:    movl   $0x0,-0xc(%rbp)
   0x00000000004004fa <+22>:    jmp    0x400506 <main+34>
   0x00000000004004fc <+24>:    mov    -0xc(%rbp),%eax
   0x00000000004004ff <+27>:    add    %eax,-0x8(%rbp)
   0x0000000000400502 <+30>:    addl   $0x1,-0xc(%rbp)
   0x0000000000400506 <+34>:    cmpl   $0x4,-0xc(%rbp)
   0x000000000040050a <+38>:    jle    0x4004fc <main+24>
   0x000000000040050c <+40>:    mov    -0x8(%rbp),%eax
   0x000000000040050f <+43>:    mov    %eax,%edi
   0x0000000000400511 <+45>:    callq  0x4004d6 <mult2>
   0x0000000000400516 <+50>:    mov    %eax,-0x4(%rbp)
   0x0000000000400519 <+53>:    mov    $0x0,%eax
   0x000000000040051e <+58>:    leaveq
   0x000000000040051f <+59>:    retq
End of assembler dump.
```

push %rbp은 스택에 값을 넣는 것을 push라 하며 메모리에 rbp2 를 main으로 잡는 역할을 한다. 위의 3번째 줄인 'sub $0x10,%rsp' 에서 RSP가 이동하면서 생기는데 생성된다. 지역 변수 스택이 위치하는 영역이다. 이는, 16진수를 10진법으로 바꾸면 16 정도의 공간이 생긴다. 여기에 밑의 rbp의 값이 들어간다. jmp 0x400506 <main+34> 구간에서 연산을 위해 레지스터로 값이 옮겨가는 것으로 보여진다. -0xc(%rbp)를 %eax에 넣어주고 그 다음 실행구간에서 서로 더하여 주는 것으로 보여진다.

10. 구구단을 만들어보시오.

```
#include <stdio.h>
int mul(int a, int b)
{
        return(a*b);
}
int main(void)
```

```
{
	int a = 1;

	while (a < 10)
	{
		int b = 1;
		while (b < 10)
		{
			printf("%d * %d = %d ₩n", a , b, mul(a,b));
			b++;
		}
		a++;
		printf("₩n");
	}
	return 0;
}
```



12. 리눅스에서 Debugging하는 방법에 대해 기술해보시오. Gdb 상에서 아직 소개하지 않은 명령 등, bt, c 이 2개에 대해 조사해보고 활용해보자.

먼저 작성한 파일을 gcc [소스파일명]을 하여 실행파일을 생성한다.

> ls 명령어로 보면 a.out이란 실행파일이 생성되어 있다.

컴파일할 때, gcc – o [바뀔 이름] 을 입력하면 컴파일시 [ ]에 컴파일 한 c 파일이 생성된다.

gcc –g –o 는 debug가 가능하도록 만든 컴파일이 생성된다.

gcc –g –o0 –o 는 최적화가 방지 된 채 생성 되도록 해준다(컴파일러가 많은 최적화를 하기 때문에 나온 옵션이다. –o0는 영문o 숫자0의 순서로 써진다).

그러고 나서 ls로 확인하면 debug가 생성되어 있다.

gdb debug (gdb [실행파일]) 를 입력하면 실행파일이 디버깅 된다. 이때 disas을 하면 기계어인 어셈블리어를 볼 수 있다.

bt는 bracktrace의 약어로 프로그램 스택을 보여준다.

c 는 브레이크포인트로 일시 정지된 프로그램을 계속 실행한다.

명령어	의미
list	현재 위치에서 원본 파일의 코드를 10줄만 보여준다. 만일 list 5, 15라고 입력하면 5번째에서 15번째 줄까지 보여준다.
run(r)	프로그램을 실행한다.
break	특정 라인이나 함수에 브레이크포인트를 설정한.
Clear	특정 라인이나 함수에 있던 브레이크포인드를 삭제한다.
Print expr	효션식의 값을 보여준다.
Next	브레이크포인터에 의해 정지된 상태에서 다음 라인을 실행한다. 이때 함수 호출은 무시한다.
Edit [file:]function	브레이크포인트에 의해 저지된 상태에서 실행 라인의 위치를 보여준다.
Step	브레이크포인트에 의해 정지된 상태에서 다음 라인을 실행한다. 이때 함수 호출이 있으면 해당함수로 이동한다.
help	Gdb 명령어의 사용법을 출력한다.
quit	gdb에서 빠져 나온다.