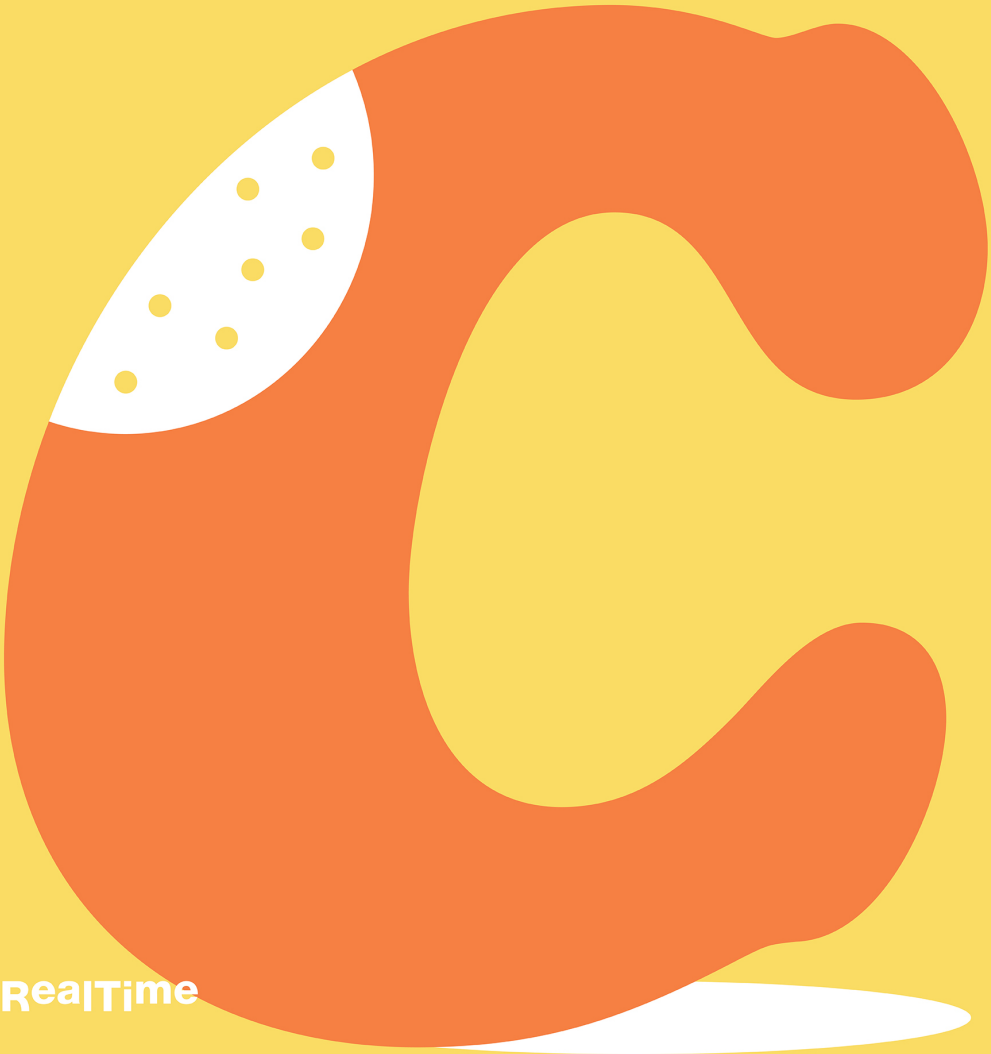
C 발라먹기

C 언어 기초 알짜배기

신일고등학교 자율동아리,
2017학년도 GIRL Is Recursion Lover
2018학년도 BOY is Object-oriented programming Yearner 지음



RealTime

들어가며

C 발라먹기는 발라먹기 시리즈의 첫 번째 책으로, 자율형 사립고 신일고등학교 2017학년 자율동아리 *GIRL Is Recursion Lover*의 동아리원의 효과적인 C 언어 학습을 위해 제작되었지만, 2018학년 자율동아리 *BOY is Object-oriented programming Yearner*를 위해 내용을 대폭 추가하고 약간의 오류를 정정했습니다.

입력과 출력과 같은 기본적인 내용부터 배열과 포인터 등에 대해서도 다루고 있으며, 잠깐 스쳐 지나가는 정도이지만 컴퓨터의 구조에 대한 약간의 정보도 담고 있습니다. 이 교재는 C 언어의 고급 내용까지 다루고 있지는 않지만, 여러분이 C 언어에 대해 충분히 체계적이고 탄탄한 기초를 쌓는 데에는 큰 도움이 될 것입니다.

이 책을 읽는 여러분이 배우려 하는 C 언어는 벨 연구소의 데니스 리치Dennis Ritchie가 만든 언어로, 그 목적은 운영체제OS: Operation System를 작성하는 데 있었습니다. C 언어로 만들어진 운영체제가 바로 리눅스Linux입니다. 물론 C 이전에도 포트란, Lisp 등의 여러 프로그래밍 언어가 있었으나, 이들은 운영체제를 만들 수 있을 만큼 충분히 섬세한 제어를 하지 못했고, 결국 C 언어가 등장하기 전까지 프로그래머는 운영체제를 만들기 위해 어셈블리 언어(기계어의 0과 1들을 묶어 영어 단어로 바꾼 것으로, 고급 언어에 비해 생산성이 매우 낮음)를 사용할 수밖에 없었습니다.

초창기의 C 언어는 어셈블리 언어보다 느렸지만, 잘 설계된 C 언어 프로그램은 그렇지 못한 어셈블리 언어보다 빠르게 동작했기 때문에 사람들은 C 언어를 사용하기 시작했습니다. CPU는 기계어밖에 읽지 못하기 때문에 C 언어 코드는 어셈블리어(기계

어)로 변환될 필요가 있는데, 이 과정을 컴파일이라 하며 이 동작을 수행하는 프로그램을 컴파일러라 합니다. 현대의 컴파일러는 무척 고속으로 작동하며, 대부분의 경우 숙련된 어셈블리어 프로그래머보다 좋은 코드를 내놓습니다.

C 언어는 다른 언어들에 비하면 어려운 축에 속합니다. 특히 C의 포인터는 넘기 어려운 벽으로 유명합니다. 하지만 C 언어가 이렇게 어려운 데는 위와 같은 이유가 있습니다. 최근에는 Go나 Rust같은 언어들이 C/C++를 대체하려 시도하지만, 아직 C의 지위는 확고합니다. C 언어는 다른 거의 모든 프로그래밍 언어에 영향을 주었습니다. 다른 프로그래밍 언어를 공부해 보면 C의 흔적들을 찾을 수 있습니다.

여러분이 프로그래밍을 공부하게 되었다는 것은 다른 그 누구보다도 인류 최신 기술 전선에 있다는 뜻입니다. 분명 어려운 부분도 있을 것이라 생각하지만, 포기하지 않고 공부하면 그 속에서 기쁨을 찾을 수 있을 것이라 믿습니다.

2018년 2월 1일

2017학년도 신일고등학교 자율동아리 *GIRL Is Recursion Lover* 동아리장,
2018학년도 신일고등학교 자율동아리 *BOY is Object-oriented programming Yearner* 동아리장

김환희

목차

컴퓨터와 소통하기

출력 1

동서고금을 떠나, 말을 배운 아기가 가장 처음 하는 말은 '엄마'라고 합니다. 믿거나 말거나. 프로그래밍 세계에서도 말하는 법을 배웠을 때 '엄마' 대신 하는 말이 있습니다. 바로 'Hello, world!'입니다. 자, 따라 해 보세요. Hello, world!

```
#include <stdio.h>
int main() {
    printf("Hello, world!\n");        //Hello, world!를 출력하고 줄 바꿈
    return 0;
}
```

> Hello, world!

C 언어는 **위에서 아래로 실행**됩니다. 자, 맨 윗줄부터 조금씩 읽어 봅시다. ***#include <stdio.h>*는 *stdio.h*를 포함**시키라는 의미입니다. C 언어에서 *stdio.h*는 **입출력**을 담당하는 헤더로, 사람으로 치면 눈과 입에 해당합니다.

그다음 줄은 ***int main()***입니다. '정수 값을 반환하고 아무런 인자도 받지 않는 함수 *main*를 정의하겠습니다'라는 의미인데, 지금은 별로 알 필요가 없습니다.

main 함수는 수많은 함수 중 가장 특별한 함수로, 모든 C 프로그램에는 ***main* 함수가 반드시 포함**되어야 하며, **프로그램이 시작되면 *main***

함수가 실행됩니다. 이 것도 뒤의 '함수' 파트에서 자세히 다룰 것입니다.

함수 안의 내용은 *{ 중괄호 }*로 둘러싸여 있습니다. 여는 괄호의 갯수와 닫는 괄호의 갯수는 반드시 일치해야 합니다.

printf* 함수는 문자열을 받아 출력**하는 함수입니다. **문자열은 *"큰 따옴표"로 둘러싸인 것입니다. 문자열의 진정한 의미는 뒤의 '포인터' 부분에서야 알 수 있지만, 일단은 현재의 코드에 집중해 봅시다. 우리는 printf 함수에 *"Hello, world!₩n"*라는 문자열을 넘겨주었는데, 이렇게 하면 printf 함수는 Hello, world!를 출력하게 됩니다. 뒤의 *₩n*은 줄을 바꾸라는 의미입니다. 이런 것을 '**이스케이프 시퀀스**'라고 하며, 다음과 같은 것들이 있습니다.

형식	의미하는 것	형식	의미하는 것
₩n	줄 바꿈	*₩'*	작은 따옴표
₩t	가로 Tab	*₩"*	큰 따옴표
₩v	세로 Tab	*₩₩*	₩
₩b	백 스페이스	*₩OOO*	8진수 아스키 코드
₩r	캐리지 리턴	*₩xOO*	16진수 아스키 코드
₩f	폼 피트	*₩xOOOO*	16진수 유니 코드
₩a	경고 벨	*%%*	*%*

읽고 넘어가기

- 환경에 따라 백 슬래시가 ₩로 보일 수 있습니다.
- //로 시작하는 내용은 **주석**으로, **컴파일러에게 읽히지 않도록 한 부분**입니다. '주석' 파트에서 자세히 다룹니다.

마지막 *return 0*은 **함수가 문제없이 실행되었음**을 나타냅니다. 진짜 의미는 '**0을 반환**'이지만, 아직 함수가 값을 반환한다는 것에 대해 알지 못하므로, '그렇구나' 하고 넘어가도록 합시다. ***main*에서 *return 0*을 만나면 프로그램은 종료**됩니다.

눈치가 빠른 사람이라면 코드의 뒤에 세미콜론;이 붙어있음을 눈치 챘을 것입니다. 함수의 실행, 값의 대입, 값의 반환(*return*) 등의 **명령어 뒤에는 반드시 세미콜론**을 사용해 줄을 구분해야만 합니다.

C 언어의 다른 특징 중 하나는 구분만 할 수 있다면 **단어와 기호들의 띄어쓰기와 줄 바꿈이 의미를 갖지 않는다**는 것입니다. **#으로 시작하는 줄은 예외**입니다. 즉,

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

이 코드도,

```
#include <stdio.h>
```

```
int main()
{
    printf(
                "Hello, world!\n"

);

                    return 0;
}
```

이 코드도, 처음의 코드와 같은 의미를 지닙니다. 하지만

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

이건 안됩니다.

많은 프로그래밍 언어들은 '**들여쓰기**'를 권장하고 있습니다. *printf* 앞의 공백이 보입니까? 그것이 들여쓰기입니다. *if*와 같은 문법을 배우게 되면 함수 안에서도 중괄호를 사용하게 될텐데, 이러한 경우에는 더 많은 공백을 사용합니다. 조금 후 코드를 보고 이해하십시오.

이러한 들여쓰기는 대부분 4회의 Space나 Tab 키를 사용합니다. 이는 IDE(통합 개발 환경)나 에디터의 설정에 따라 다르며, 대부분의 IDE/

에디터들은 엔터를 칠 때마다 자동으로 들여쓰기를 해 줍니다. 이 책의 모든 코드는 6회의 띄어쓰기 Space를 사용했습니다.

마지막으로 자신의 이름을 출력하고, 이 챕터를 마치겠습니다.

```
#include <stdio.h>
int main() {
	printf("Hello, BOY!\n");		//Hello, BOY!를 출력하고 줄바꿈
	return 0;
}
```

> Hello, BOY!

그리고 조금 미안한 말이지만, 앞으로도 포인터를 모르면 정확한 의미를 알 수 없는 말들이 나올 수 있습니다. 이런 말 뒤에는 ★을 붙일 테니, 지금은 건너뛰었다가 나중에 포인터를 배운 후 다시 와서 읽어보세요.

주석

프로그래밍에서 **주석**은 **컴파일러에게 읽히지 않도록 한 부분**입니다. 절대 시진핑같은 사람이나 금속을 가리키는 말이 아닙니다. 여기서 컴파일러는 C언어 코드를 기계어로 번역해 주는 프로그램을 말합니다.

주석은 크게 2가지의 사용 방법이 있습니다. 첫 번째는 //입니다.

```
#include <stdio.h>
int main() {
	//printf("그아아앗\n");
	printf("퉤에에엣\n");	//착한 어린이가 됩시다.
	return 0;
}
```

> 퉤에에엣

//가 있는 줄의 // 이후 부분은 모두 주석입니다. 이 코드를 실행해 보면, *퉤에에엣*만이 출력되는 것을 볼 수 있습니다. 또, 두 번째 주석의 한글은 C 언어 문법에 맞지 않는데, 주석이기 때문에 상관없이 실행됩니다.

두 번째 주석은 /* */입니다.

```
#include <stdio.h>
int main() {
	/*printf("그아아앗\n");
```

```
    printf("퉤에에엣\n");*/
    printf("삐야아악\n");
    return 0;
}
```

> 삐야아악

이번에는 *삐야아악*만이 출력됩니다. **/*와 */ 사이의 내용이 주석으로 처리**되기 때문입니다. 하지만 이런 것은 안됩니다.

```
#include <stdio.h>
int main() {
    /*printf("그아아앗\n");
    /*printf("퉤에에엣\n");*/
    printf("삐야아악\n");*/
    printf("뿌우우웅\n");
    return 0;
}
```

왜냐하면 컴파일러가 첫 번째 /*과 첫 번째 */를 짝으로 인식해 두 번째 /*는 주석 속에 묻히고, 결국 마지막 */만이 덩그러니 남는데, **짝이 맞지 않는 */는 문법 오류** 사항이기 때문입니다.

주석은 대개 코드에 대해 **설명**을 하거나, 어느 부분에서 오류가 나는지 알아내기 위해 사용합니다. 다만, 지나친 주석은 코드의 가독성을 해치거나, 주석에 코드가 끌려다니는 결과를 낳게 될 수도 있으니 주의해서 사용하십시오.

자료형

자료형Type**은 데이터들의 분류**로, 같은 성질을 갖는 값들의 집합을 말합니다. C언어에서 기본적으로 제공하는 자료형들은 메모리에서 차지하는 크기와 소수점 유무에 따라 분류되며, *char*, *short*, *long* 등이 있습니다. 각각 1byte, 2byte, 4byte의 공간을 차지합니다.

대표적으로 **정수 자료형**은 다음과 같습니다.

char	1byte	$-2^3 \sim 2^3 - 1$
unsigned char	1byte	$0 \sim 2^4 - 1$
short	2byte	$-2^7 \sim 2^7 - 1$
unsigned short	2byte	$0 \sim 2^8 - 1$
long	4byte	$-2^{15} \sim 2^{15} - 1$
unsigned long	4byte	$0 \sim 2^{16} - 1$
long long	8byte	$-2^{31} \sim 2^{31} - 1$
unsigned long long	8byte	$0 \sim 2^{32} - 1$
	32bit에서 4byte, 64bit	$-2^{15} \sim 2^{15} - 1$, -2^{31}

int	에서 8byte	$\sim 2^{31}-1$
unsigned int	32bit에서 4byte, 64bit 에서 8byte	$0 \sim 2^{16}-1$, $0 \sim 2^{32}-1$

잘 보면, ***unsigned* 키워드를 붙여 0을 포함한 양수**만을 사용할 수 있습니다. 또, *int*의 크기는 환경에 따라 다른데, 이는 *int*가 '실행되는 환경에서 가장 빠른 크기'로 정의되어 있기 때문입니다. 우리가 '32bit 컴퓨터', '64bit 컴퓨터' 하는 것은 '그 컴퓨터에서 명령어가 얼만큼의 길이를 갖는지'를 나타내는데, 명령어의 길이와 다른 데이터를 처리할 때는 일단 데이터를 명령어의 길이에 맞추는 과정이 필요하기 때문에 명령어와 같은 길이를 갖는 데이터, 즉 4byte 혹은 8byte의 데이터만이 그러한 과정 없이 빠르게 실행될 수 있습니다. 이 책에서는 주로 *int*을 사용할 것입니다.

또, 소수점이 있는 수를 나타내기 위한 자료형인 **실수 자료형**도 있습니다.

float	4byte	16bit 정밀도
double	8byte	32bit 정밀도
long double	12byte	48bit 정밀도

이들 실수 자료형은 **부동 소수점** 방식을 사용합니다. **떠돌이 소수점**이라는 말로도 부릅니다. 부동 소수점 방식은 일단 소수점이 어디에 있는

지 상관없이 수를 적은 후, 나중에 소수점의 위치를 적는 방식입니다. 즉, 최대·최소 크기보다 **최대 자릿수**가 중요합니다.

나중에는 이 자료형들을 묶어 만든 **구조체**나 **포인터★**와 같은 유도 자료형들도 등장하나, 여기에서는 이만큼만 알아두도록 합시다.

변수와 상수, 리터럴

변수는 **값을 담아두는 공간**입니다. ***자료형 변수 이름***의 형식으로 선언합니다. **각 변수는 메모리상의 공간을 차지하고 있습니다.**

```
int main() {
    int a;     //int형 변수 a
    char b, c;  //char형 변수 b, c
    return 0;
}
```

변수의 이름에는 규칙이 있습니다. 변수명에는 **알파벳 문자와 숫자, _**를 사용할 수 있으며, 중간에 **공백을 사용할 수 없습니다**. 또한, **첫 번째 문자는 알파벳 또는 _**여야 합니다. 알파벳 **대문자와 소문자를 구별**하며, C에서 이미 사용되고 있는 이름인 **예약어**(*if*, *for* 등)**는 변수명으로 쓸 수 없습니다**.

초기화는 변수의 선언과 동시에 값을 담는 것을 말합니다. 만일 초기화를 하지 않는다면, **변수에는 랜덤한 값(쓰레기 값)**이 들어가 있을 것입니다. 따라서 가능하다면 반드시 초기화하는 것이 좋습니다.

```
int main() {
    int a = 5;      //a를 5로 초기화
```

```
    int b = 3, c = b; //b를 3으로 초기화, c를 b(3)으로 초기화
    return 0;
}
```

대입은 변수에 값을 담는 것을 말합니다.

```
int main() {
    int a = 5;        //a를 5로 초기화
    int b = 3, c = b; //b를 3으로 초기화, c를 b(3)으로 초기화
    a = 2;            //a에 2 대입
    b = a + 3;        //b에 a + 3, 즉 5 대입
    c = 1;            //c에 1 대입
    return 0;
}
```

const 키워드를 사용해 상수를 선언할 수 있습니다. 상수의 값은 바꿀 수 없습니다.

```
int main() {
    const int a = 5;  //a를 5로 초기화
    int b = 3, c = b; //b를 3으로 초기화, c를 b(3)으로 초기화
    a = 2;            //a에 2 대입, 오류!
    return 0;
}
```

아무 곳에나 대입할 수는 없습니다. **=을 기준으로 왼쪽에 올 수 있는 값을 *lvalue*, 오른쪽에 올 수 있는 값을 *rvalue***라고 합니다. *lvalue*는 메모리상의 **공간을 차지**하고 있으며, 후에 배우겠지만 *&*연산자를 사용해 **주소를 가져올 수 있습니다★**. ***lvalue*는 = 오른쪽에 쓸 수 있지만 *rvalue*는 = 왼쪽에 쓸 수 없습니다**. *3*, *-5.6*, *'k'*, *"Hello, world!₩n"*와 같은 리터럴들은 *rvalue*가 될 수 있지만 *lvalue*는 될 수 없습니다. 참고로, 문자열 리터럴은 사실 주소값입니다★.

```
int main() {
    int a = 5, b = 3; //a를 5로 초기화, b를 3으로 초기화
    a = 2;          //a에 2 대입, a는 lvalue, 2는 rvalue
    b = a + 1;      //b에 a + 1, 즉 3 대입, b는 lvalue, a + 1은 rvalue
    b + 1 = 5;      //b + 1에 5 대입, b + 1은 rvalue, 5는 rvalue, 오류!
    return 0;
}
```

연산자

C는 다양한 연산자를 제공하고 있습니다. 각 연산자는 **우선순위**와 **결합 방향**을 가지고 있습니다. C++처럼 연산자를 프로그래머가 재지정할 수 있는 언어도 있지만, C는 그렇지 못합니다. 다음은 연산자들의 목록입니다. 값이 바뀌는 연산자는 굵게 표시했으며, *lvalue*라고 표시하지 않은 모든 연산의 결과는 *rvalue*입니다.

한 가지 알아두자면, 이름에 '논리'가 들어간 연산에서 **'참'은 *0*이 아닌 값(주로 *1*)**을, **거짓은 *0***을 나타냅니다. 이는 뒤의 *if*나 *for* 등에서도 사용하니 알아두세요.

1순위, 좌에서 우로 결합

멤버 선택	*a.b*	*a*의 멤버 *b*
멤버 선택	*a->b*	포인터 *a*가 가리키는 값의 멤버 *b*, *(*a).b*와 같은 뜻★
배열 첨자	*a[b]*	배열 *a*의 *b*번째 원소, **(a + b)*와 같은 뜻★, *lvalue*
함수 호출	*a(b, c)*	함수 *a*에 인자 *b, c* 전달, *rvalue* 혹은 값 없음
후위 증		*a*의 복사본(*rvalue*)을 *a++*의 자리에 놓고 *a*자신

가	*a++*	은 1 증가
후위 감소	*a--*	*a*의 복사본(*rvalue*)을 *a--*의 자리에 놓고 *a*자신은 1 감소

2순위, 우에서 좌로 결합

크기 측정	*sizeof(a), sizeof a, sizeof(T)*	*a*가 차지하는 크기, 타입 *T*가 차지하는 크기
전위 증가	*++a*	*a*가 *1* 증가, 증가한 자기 자신(*lvalue*)
전위 감소	*--a*	*a*가 1 감소, 감소한 자기 자신(*lvalue*)
비트 NOT	*~a*	*a*에 대한 비트 NOT(1의 보수)
논리 NOT	*!a*	*a*에 대한 논리 NOT
단항 덧셈	*+a*	*a* 자기 자신
단항 뺄셈	*-a*	*a*에 대한 부호 바꾸기
주소	*&a*	*a*의 주소
참조	**a*	주소 *a*가 가리키는 값, *lvalue*

캐스트	*(T)a*	*a*를 *T*타입으로 캐스트

3순위, 좌에서 우로 결합

곱셈	*a * b*	*a*와 *b*의 곱
나눗셈	*a / b*	*a*를 *b*로 나눈 것
나머지	*a % b*	*a*를 *b*로 나눈 나머지

4순위, 좌에서 우로 결합

덧셈	*a + b*	*a*와 *b*의 합
뺄셈	*a − b*	*a*와 *b*의 차

5순위, 좌에서 우로 결합

왼쪽 시프트	*a 〈〈 b*	*a*를 좌로 *b*만큼 시프트 한 것, 부호 없는 정수형에 한해 $a * 2^{b-1}$과 같은 뜻
오른쪽 시프트	*a 〉〉 b*	*a*를 우로 *b*만큼 시프트 한 것, 부호 없는 정수형에 한해 $a / 2^{b-1}$과 같은 뜻

6순위, 좌에서 우로 결합

작음	a < b	a가 b보다 작다면 참, 그렇지 않다면 거짓
큼	a > b	a가 b보다 크다면 참, 그렇지 않다면 거짓
작거나 같음	a <= b	a가 b보다 작거나 같다면 참, 그렇지 않다면 거짓
크거나 같음	a >= b	a가 b보다 크거나 같다면 참, 그렇지 않다면 거짓

7순위, 좌에서 우로 결합

같음	a == b	a와 b가 같다면 참, 그렇지 않다면 거짓
같지 않음	a != b	a와 b가 같지 않다면 참, 그렇지 않다면 거짓

8순위, 좌에서 우로 결합

비트 AND	a & b	a와 b의 비트 AND

9순위, 좌에서 우로 결합

비트 XOR	a ^ b	a와 b의 비트 XOR

10순위, 좌에서 우로 결합

비트 OR	a | b	a와 b의 비트 OR

11순위, 좌에서 우로 결합

논리 AND	*a && b*	*a*와 *b*의 논리 AND

12순위, 좌에서 우로 결합

논리 OR	*a || b*	*a*와 *b*의 논리 OR

13순위, 우에서 좌로 결합

조건 삼항	*a ? b : c*	*a*가 참이라면 *b*, 그렇지 않다면 *c*

14순위, 우에서 좌로 결합

대입	***a = b***	*a*에 *b* 대입, *a*는 상수가 아닌 *lvalue*
연산 후 대입	***a += b, a -= b, a *= b...***	각각 *a = a + b, a = a - b, a = a * b...*와 같은 뜻

15순위, 좌에서 우로 결합

쉼표	*a, b*	*a* 명령어 실행 후 *b*명령어 실행

아직 배우지 않은 용어들이 나와 익숙하지 않을 것입니다. 증감 연산 및 사칙 연산, 나머지, 대입, 연산 후 대입 정도만 알아두십시오.

출력 2

자, 변수를 선언하고 값을 담는 법을 알았으니, 이제 그 수를 출력하는 법에 대해 알아봅시다.

```
#include <stdio.h>
int main() {
    int a = 5, b = 3;
    printf("a는 %d, b는 %d, a + b는 %d\n", a, b, a + b);
    return 0;
}
```

> a는 5, b는 3, a + b는 8

여기서 *%d*는 **'4바이트 부호 있는 정수'가 *%d*의 자리에 옴**을 의미합니다. 보면 문자열 뒤로 *a, b, a + b*를 *printf*에 넘겨주는 모습을 볼 수 있는데, 각각 첫 번째 *%d*, 두 번째 *%d*, 세 번째 *%d*로 출력됩니다. 여기서의 *%d*를 **형식 문자**라고 합니다.

다음은 자주 사용되는 형식 문자입니다.

%d	부호 있는 10진수 정수
%u	부호 없는 10진수 정수
%o	부호 없는 8진수 정수

%c	값에 해당하는 아스키 문자
%x	부호 없는 16진수 정수, 소문자 사용
%X	부호 없는 16진수 정수, 대문자 사용
%f	실수 자료형 소수점 아래 6자리 출력, 다른 자릿수는 *%.자릿수f*
%s	문자열 출력, *₩0*이 나올 때까지 1바이트씩 출력하며 주소값 받음

*%s*에 대해서는 당장 이해하지 못해도 좋습니다.

여기에 다음의 **접두사**를 사용해 확장합니다.

l	*long int, unsigned long int*
ll	*long long*
h	*short int, unsigned short int*
l	*long double*

접두사는 '*%llu*가 부호 없는 8자리 정수', '*%.8lf*가 *long double* 소수점 아래 8자리'와 같이 사용합니다. 물론 가장 많이 사용하게 될 것은 ***%d*** 입니다.

```
#include 〈stdio.h〉
int main() {
```

```
    int a = 5;
    unsigned short int b = 0xFFFF;//0x으로 시작하면 16진수
    char c = 'S';           //S의 아스키 코드값은 83, 사실 'S'는 1바이트 정수
83이다
    printf("a는 %d\n", a);
    printf("b는 %hu...16진수 %hX\n", b, b);
    printf("c는 %c...아스키 코드값 %d", c, c);
    return 0;
}
```

> a는 5
> b는 65535...16진수 FFFF
>
> c는 S...아스키 코드값 83

printf 이외에도 여러 출력 수단이 있습니다. ***puts***와 ***putchar***이 그것입니다.

```
#include <stdio.h>
int main() {
    puts("그아아앗");
    puts("퉤에에엣");
    putchar(66);      //B의 아스키 코드값은 66
    putchar('O');
    putchar('Y');
    return 0;
}
```

> 그아아앗
> 퉤에에엣
>
> BOY

*puts*는 **문자열을 출력하고, 한 줄 띕니다**. 즉, 위의 *puts("퉤에에엣")*은 *printf("퉤에에엣₩n")*과 같은 것을 출력합니다. *putchar*는 말 그대로 **문자 한 개를 출력**합니다. 따옴표로 둘러싸인 *'B'*를 보내던, B의 아스키 코드값인 *66*을 보내던 똑같이 동작합니다. 다만 저 ***66*은 4바이트지만 *'B'*는 1바이트**라는 것을 알아두세요.

입력

***scanf* 함수를 사용해 값을 입력**받을 수도 있습니다. *scanf* 함수의 사용법은 *printf* 함수의 사용법과 아주 유사합니다.

```
#include <stdio.h>
int main() {
    int a;
    scanf("%d", &a);                //절대 &를 빠뜨리지 마세요!
    printf("입력받은 값은 %d입니다.\n", a);
    return 0;
}
```

실행하게 되면 콘솔 창에 아무 것도 나타나지 않을 것입니다. 자, 아무 수나 입력하고 엔터를 눌러줍시다. 한 번 4를 입력해 볼까요?

> 입력받은 값은 4입니다.

이렇게 *scanf* 함수를 사용해 **키보드(표준 입력)로부터 값을 입력**받을 수 있습니다. 하지만 주의하세요! 저 ***&*기호를 빼먹으면 이상한 결과가 나오거나 프로그램이 강제로 종료될 수 있습니다.**

```
#include <stdio.h>
int main() {
    int a, b;
```

```
    scanf("add%dto%d", &a, &b);
    printf("%d + %d = %d\n", a, b, a + b);
    return 0;
}
```

> add5to3

> 5 + 3 = 8

*printf*처럼 **입력받는 수의 형태를 정해놓을 수도 있습니다**. 문자열의 입력과 출력에 관한 부분은 '문자열' 파트에서 다시 다루도록 하겠습니다.

컴퓨터와 친구하기

명시적 형변환

(원하는 타입)값의 형식으로 한 값의 자료형을 바꿔줄 수 있습니다.

```
#include <stdio.h>
int main() {
	float a = 6.5;
	int b = (int)a;
	printf("a: %.1f\n", a);
	printf("b: %d\n", b);
	return 0;
}
```

> a: 6.5
>
> b: 6

사실, 이런 상황에서는 *(int)*를 쓰지 않아도 **자동으로 바꿔주지만**, 이것은 '**묵시적 형변환**'에서 다룰 것입니다. 묵시적인 형변환이 일어나는 상황에서는 **정해진 대로**만 자료형이 바뀌기 때문에, 좀 더 확실하게, '명시적으로' 자료형을 바꿔주고 싶다면 명시적 형변환을 해 주세요. 명시적 형변환은 포인터를 사용할 때 그 진가를 발휘하니, 기대해도 좋습니다.

명시적 형변환을 하면, **값이 손실**될 수도 있습니다.

```
#include <stdio.h>
int main() {
    printf("%hX\n", (short)0x12345678);
    return 0;
}
```

> 5678

사실 저 코드에서 *%hX*를 *%X*로 바꾸거나, *(short)*를 빼도 같은 결과를 얻을 수 있습니다. 하지만 둘 다 빼는 것은 안됩니다. *printf*가 인자를 받는 방식(**가변 인자**)에 대해 공부해 보면 그 이유를 알 수 있습니다만, 여기서는 다루지 않았으니 일단은 '그렇구나' 하고 넘어가시고, 책을 모두 읽은 다음 스스로 공부해 보세요.

배열

배열은 같은 이름으로 변수 묶음을 만든 것입니다. ***자료형 변수명[크기]***의 형식으로 선언하며, ***0번째부터 크기 - 1번째까지 사용***할 수 있습니다. **중괄호를 사용해 초기화**하며, 초기화할 경우 **크기를 생략**할 수 있습니다(자동으로 원소의 개수에 맞추어 초기화됩니다). ***{ 0, } 혹은 { 0 }으로 초기화할 경우 모든 원소가 0***이 되며, 이 경우 크기를 생략할 수 없습니다.

```
#include <stdio.h>
int main() {
    int a[ 10 ] = { 0, 2, 4, 6, 8, 10 },// 10 뒤의 값들은 쓰레기 값
        b[ 10 ] = { 0, },              //모두 다 0으로 초기화, { 0 }도 가능
        c[]     = { 1, 3, 5 }, d = 2;
    printf("a[ 0 ]: %d\n", a[ 0 ]);
    printf("a[ d ]: %d\n", a[ d ]);
    printf("b[ 0 ]: %d, b[ 0 ]: %d...\n", b[ 0 ], b[ 1 ]);
    printf("c[ 0 ]: %d, 메모리상에서 %d바이트를 쓰고 있으므로 길이는 %d",
c[ 0 ],
    sizeof c, sizeof c / sizeof(int));
    return 0;
}
```

```
a[ 0 ]: 0
a[ d ]: 4
```

> b[0]: 0, b[0]: 0...
> c[0]: 1, 메모리상에서 12바이트를 쓰고 있으므로 길이는 3

sizeof *c*를 통해 배열 *c*가 차지하는 전체 크기를 바이트 단위로 알 수 있고, *sizeof(int)*로 나누어 길이를 알 수 있습니다.

배열의 크기를 벗어난 곳을 참조하면 어떻게 될까요?

```
#include <stdio.h>
int main() {
    int a[ 2 ] = { 1, 2 }, b[ 2 ] = { 3, 4 }, c[ 2 ] = { 5, 6 };
    printf("%d %d %d %d %d %d\n", b[ -2 ], b[ -1 ], b[ 0 ], b[ 1 ], b[ 2 ], b[ 3 ]);
    return 0;
}
```

> 0 0 3 4 0 0

사실 가운데 *3 4* 부분을 제외하면 여러분의 환경에서도 같은 결과가 나오라는 보장을 할 수 없습니다. 저는 저 자리에 **우연히** *0*이 들어있었던 것뿐이며, 운이 없다면 *-3967 98 3 4 124872342 -234857*같은 결과도 가능합니다. 심지어, 만약 컴파일러가 이런 저런 최적화들을 해 주지 않았더라면 *1 2 3 4 5 6*이 나올 수도 있었습니다★! 따라서 아주 특수한 상황이 아닌 이상 **크기를 벗어난 곳을 참조하지 마십시오**. 뒤의

'포인터' 부분에서 *[]*가 진짜로 의미하는 것이 무엇인지 설명하겠습니다.

배열의 배열을 만드는 것도 가능합니다!

```
#include <stdio.h>
int main() {
    int a[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
    printf("%d %d %d %d\n", a[ 0 ][ 0 ], a[ 0 ][ 1 ], a[ 1 ][ 0 ], a[ 1 ][ 1
]);
    return 0;
}
```

> 1 2 3 4

단, **크기 생략은 맨 앞의 것만 가능**합니다.

```
int main() {
    int a[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } }, //OK
        b[][ 2 ]    = { { 1, 2 }, { 3, 4 } }, //OK
        c[ 2 ][]    = { { 1, 2 }, { 3, 4 } }; //NO!
        d[][]       = { { 1, 2 }, { 3, 4 } }; //NO!
    return 0;
}
```

이런식으로 배열의 배열의 배열, 배열의 배열의 배열의 배열...을 만드는 것도 가능합니다. ***n*차원 배열**이라고 부릅니다. **다차원 배열의 모든 원소는 사실 메모리 상에서 일직선상**에 위치해 있습니다★.

```
int main() {
	int a[ 2 ][ 2 ][ 2 ][ 2 ][ 2 ][ 2 ] = { 0, };
	return 0;
}
```

다차원 배열에서도 이런 짓은 금지입니다.

```
#include <stdio.h>
int main() {
	int a[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
	printf("%d %d %d %d\n", a[ 1 ][ -2 ], a[ 1 ][ -1 ], a[ 1 ][ 0 ], a[ 1 ][
1 ]);
	return 0;
}
```

> 1 2 3 4

이건 제가 운이 좋아서 *1 2 3 4*가 나온 것으로, 원래 이런 짓은 하면 안 됩니다. 인덱스에 음수를 사용하는 것은 '불법적인' 행동이며, 따라서 컴파일러의 최적화에 의해 *a[0]*의 *{ 1, 2 }*가 사라졌다고 해도 여러분은 따질 곳이 없습니다.

뒤의 '포인터' 부분에서 배열에 대해 좀 더 심화적으로 학습할 것입니다.

문자열

지금껏 문자열에 관련된 내용들은 어물쩍 넘어갔습니다. *"BOY"*와 같은 것 말이죠. 왜냐하면, **문자열은 하나의 자료형이 아니라 그저 *char*의 배열**이기 때문입니다. 하지만 그냥 *char* 배열과는 다른 점이 있는데, 그것은 **마지막에 반드시 '₩0'(*NULL*)이 들어간다는 것입니다.** ***printf*도, *puts*도 '₩0'이 나올 때까지 출력**합니다. 눈치가 빠른 사람이라면, **'₩0'이 그냥 숫자 *0*을 의미**함을 알았을 것입니다.

문자열을 저장하는 방법은 다음과 같습니다.

```
#include <stdio.h>
int main() {
    char a[ 4 ]   = "BOY",    //반드시 \0(NULL)을 위한 자리가 필요
         b[ 3 ]   = "HO!",    //위험합니다!
         c[]      = "GIRL",   //크기는 5
         d[]      = { 'H', 'A', 'H', 'A' },        //위험합니다!
         e[]      = { 'H', 'O', 'H', 'O', '\0' },  //OK!
         * f      = "그아아앗"; //이것들은
    char* g       = "퉤에에엣"; //의미가 조금 다릅니다
    char* h[ 2 ]  = { "nya", "ayn" };   //이것도 단순한 문자열의 배열은 아닙니다
    char i[][ 5 ] = { "asdf", "fdsa" }; //이건 문자열의 배열입니다
    printf("a: %s\n", a); printf("b: %s\n", b);
    printf("c: %s, 크기는 %d\n", c, sizeof c / sizeof(char));
    printf("d: %s\n", d); printf("e: %s\n", e);
```

```
    printf("f: %s\n", f); printf("g: %s\n", g);
    printf("h[ 0 ]: %s\n", h[ 0 ]);
    printf("h[ 1 ]: %s\n", h[ 1 ]);
    printf("i[ 0 ]: %s\n", i[ 0 ]);
    printf("i[ 1 ]: %s\n", i[ 1 ]);
    return 0;
}
```

```
a: BOY
b: HO!BOY

c: GIRL, 크기는 5
d: HAHAGIRL

e: HOHO
f: 그아아앗

g: 퉤에에엣
h[ 0 ]: nya

h[ 1 ]: ayn
i[ 0 ]: asdf

i[ 1 ]: fdsa
```

다소 깁니다만, 천천히 음미해 주세요.

*a*는 크기 *4*짜리 *char* 배열입니다. 만약 크기를 *3*으로 했더라면 *'₩0'*(*NULL*)이 들어갈 공간이 없어 출력할 때 애로사항이 생겼을 것입니다 또, *= "BOY"*는 *= { 'B', 'O', 'Y', 0 }*와 같은 의미입니다.

*b*는 크기 *3*짜리 배열입니다. *printf*는 *'₩0'*을 만날 때까지 출력하는데, *b*의 끝에 *'₩0'*이 없기 때문에 엉뚱한 것까지 출력합니다. 여기에서는 *b[3]*이 되었어야 하는 공간이 우연히 *a[0]*과 같은 탓에 *BOY*까지 출력되었고, *a[3]*에 있는 *'₩0'*을 만나 겨우 출력을 멈추었습니다★.

*c*는 크기를 정하지 않았습니다. *GIRL*은 *4*글자이므로, *₩0*까지 더해져 총 *5*의 크기를 갖게 됩니다.

*d*는 말 그대로 *char* 배열입니다. 그런데, *'₩0'*이 없습니다. 여기에서는 *d[4]*이 되었어야 하는 공간이 우연히 *c[0]*과 같은 탓에 *GIRL*까지 출력되었고, *c[4]*에 있는 *'₩0'*을 만나 겨우 출력을 멈추었습니다★.

*e*는 *d*가 잘못한 일을 완벽히 고쳤습니다. *₩0'* 대신 그냥 *0*을 써도 괜찮습니다.

*f*와 *g*는 앞의 것들과는 조금 의미가 다릅니다★. 이 둘은 포인터 변수로, 지금 당장 알 필요는 없습니다. 이렇게도 문자열을 저장할 수 있다는 것만 알아두세요.

*h*도 문자열들의 배열이라기보다는 *char* 포인터들의 배열입니다★. 이것도 '포인터' 부분을 읽으면 이해할 수 있습니다.

*i*야말로 진정한 문자열의 배열입니다.

문자열을 입력받아 보겠습니다.

```
#include <stdio.h>
int main() {
	char a[ 5 ] = { 65, 65, 65, 65, 65 };  //65는 대문자 A입니다
	scanf("%s", a);
	printf("%s\n", a);
	printf("%d %d %d %d %d\n", a[ 0 ], a[ 1 ], a[ 2 ], a[ 3 ], a[ 4 ]);
	return 0;
}
```

> BBB

> BBB
>
> 66 66 66 0 65

우리는 여기서 두 가지 사실을 눈치 채야 합니다. 첫 번째는 ***%s*로 문자열을 입력받을 때는 *&*가 필요하지 않다**는 것입니다. 사실 ***%s*는 주소값을 필요로 하는데, *a* 자체가 주소값**이기 때문입니다★. '포인터' 부분에서 자세히 다룹니다. 자꾸 '포인터'거린다고 불평하지 마세요. 배열과 포인터는 떼려야 뗄 수 없는 관계입니다.

두 번째 사실은 입력받고 나면 자동으로 *'₩0'*까지 채워진다는 사실입니다. 그렇기 때문에 우리는 *입력받을 문자열 길이 + 1* 이상의 크기를 갖는 배열을 만들어 놓아야 합니다.

큰 따옴표로 둘러싸인 문자열의 실체와 아까 의미가 다르다고 했던 *f*, *g*, *h*의 진짜 의미는 '포인터' 부분에서 모두 해명하도록 하겠습니다. 진짜로!

if

C 언어에서는 ***if*를 사용해 조건 분기**를 구현할 수 있습니다. 즉, **특정 코드를 조건이 맞을 때만 실행**되게 할 수 있다는 것입니다. ***if(조건식) { 내용 }***의 형식으로 사용하며, 내용은 **조건식의 내용이 참**(*0*이 아님)**일 때만 실행**됩니다.

```
#include <stdio.h>
int main() {
	int a;
	scanf("%d", &a);
	if(a % 2 == 1) {	//==기호와 = 기호를 헷갈리지 마세요!
		printf("홀수입니다.\n");
	}
	printf("프로그램이 종료되었습니다.\n");
	return 0;
}
```

> 4
>
> 프로그램이 종료되었습니다.

> 5
>
> 홀수입니다.

> 프로그램이 종료되었습니다.

_if_의 조건에 맞지 않는 것들은 _else_에서 실행됩니다.

```
#include <stdio.h>
int main() {
	int a;
	scanf("%d", &a);
	if(a % 2 == 1) {
		printf("홀수입니다.\n");
	} else {
		printf("짝수입니다.\n");
	}
	return 0;
}
```

> 4

> 짝수입니다.

> 5

> 홀수입니다.

조건에 맞지 않는 것들 중 또 새로운 조건을 만족하는 것을 찾기 위해 _else if_를 사용합니다.

```
#include <stdio.h>
int main() {
    int a;
    scanf("%d", &a);
    if(a == 0) {
        printf("0입니다.\n");
    } else if(a % 2 == 1) {
        printf("홀수입니다.\n");
    } else {
        printf("짝수입니다.\n");
    }
    return 0;
}
```

> 0
>
> 0입니다.

> 4
>
> 짝수입니다.

> 5
>
> 홀수입니다.

*else if*는 여러 개를 이어서 쓸 수도 있습니다.

```
#include <stdio.h>
int main() {
    int a;
    scanf("%d", &a);
    if(a == 0) {
        printf("0입니다.\n");
    } else if(a == 1) {
        printf("1입니다.\n");
    } else if(a % 2 == 1) {
        printf("홀수입니다.\n");
    } else {
        printf("짝수입니다.\n");
    }
    return 0;
}
```

> 0
>
> 0입니다.

> 1
>
> 1입니다.

> 4

> 짝수입니다.

> 5

> 홀수입니다.

***else if*와 *else*는 필수 사항이 아닙니다**. 다만, **반드시 *if* – *else if* – *else* 순으로 사용**해야 합니다. 또, **중괄호 안의 내용이 단 한 줄이라면, 중괄호를 생략**할 수 있습니다. 아래 코드는 위의 코드와 같은 뜻입니다.

```
#include <stdio.h>
int main() {
    int a;
    scanf("%d", &a);
    if(a == 0)
        printf("0입니다.\n");
    else if(a == 1)
        printf("1입니다.\n");
    else if(a % 2 == 1)
        printf("홀수입니다.\n");
    else  printf("짝수입니다.\n");  //공백은 의미가 없습니다.
    return 0;
}
```

레이블과 goto

***이름:*의 형식으로 레이블을 만들 수 있습니다.**

```
int main() {
lable1:
lable2:
lable3:
    return 0;
}
```

레이블 선언문은 다른 명령어들보다 **들여쓰기를 한 단계 낮추는 것이 관례**입니다.

레이블 그 자체로는 별 의미가 없고, 레이블은 *goto*와 함께 사용할 때 비로소 제 힘을 발합니다. ***goto 레이블***의 형식을 사용해 정해진 레이블로 점프할 수 있습니다.

```
#include <stdio.h>
int main() {
    printf("START!\n");
    goto lable1;
    printf("퉤에에엣\n");
lable1:
    printf("END\n");
    return 0;
}
```

```
START!

END
```

우리가 *goto*를 사용해 *lable1*로 건너뛰었기 때문에 중간의 *퉤에에엣*은 출력되지 않습니다. 이번엔 위로 점프해 보도록 하겠습니다.

```
#include <stdio.h>
int main() {
lable1:
    printf("퉤에에엣\n");
    goto lable1;
    return 0;
}
```

```
퉤에에엣

퉤에에엣

퉤에에엣

퉤에에엣

...
```

무수히 많은 *퉤에에엣*이 쏟아져 나올 것입니다. 빨리 창을 닫으세요.

이렇게 *goto*을 사용하면 **특정 동작을 반복**할 수 있습니다. 하지만 이렇게 주구장창 *퉤에에엣*만 쏟아내고 있는 것은 의미가 없으니, 이번에는 사용자가 명령한 횟수만큼만 *퉤에에엣*을 출력하도록 하겠습니다.

```
#include 〈stdio.h〉
int main() {
	int a, i = 0;
	scanf("%d", &a);
lable1:
	printf("퉤에에엣\n");
	printf("i: %d\n", i);
	++i;  //사실 printf("i: %d\n", i++);로 했어도 같은 의미였습니다...
	if(i 〈 a)
		goto lable1;
	return 0;
}
```

> 4

> 퉤에에엣
> i: 0
>
> 퉤에에엣
> i: 1

> 퉤에에엣
> i: 2
>
> 퉤에에엣
> i: 3

딱 4회만 출력됨을 확인할 수 있습니다. 잘 보면 지금이 몇 바퀴째인지에 따라 i 값이 변하고 있습니다. 즉, 몇 바퀴 돌았는지, 그 정보를 *i*에 저장하고 *a*와 비교하며 원하는 횟수만큼만 *퉤에에엣*을 출력하고 있습니다.

GCC 컴파일러 한정으로, 레이블을 변수처럼 다룰 수 있습니다.

```
#include <stdio.h>
int main() {
    void* a[ 3 ] = { &&l1, &&l2, &&l3 };//★
    int count = 0;
START:
    goto *a[ count ];                    //★
l1:
    printf("이곳은 l1\n");
    goto END;
l2:
    printf("이곳은 l2\n");
    goto END;
l3:
```

```
    printf("이곳은 l3\n");
    goto END;
END:
    ++count;
    if(count < 3)    //이것도 if(++count < 3)로 줄일 수 있습니다.
        goto START;
    printf("END\n");
    return 0;
}
```

> 이곳은 l1
> 이곳은 l2
>
> 이곳은 l3
> END

*의 의미는 포인터를 배운 후에 알려드리겠습니다. 지금은 '이런 것이 있다' 정도만 알아두세요.

for

*if*와 *goto*만을 사용해 루프를 제어하는 것은 아주 힘든 일입니다. 놀랍게도, 구석기 시대 프로그래밍 언어인 BASIC이나 어셈블리 언어에서는 이런 것들만 사용해서 루프를 제어했다고 합니다(요즘 BASIC에는 있는 모양입니다만). 그러나 애석하게도, 우리는 이런 것에 쏟을 시간이 없습니다. 그래서 나온 것이 *for*입니다.

*for*를 사용하면 **루프**를 쉽게 만들 수 있습니다. ***for(초기화; 조건식; 증감문) { 내용 }*의 형식으로 *for*문을 만들 수 있습니다. 초기화에서는 루프 안에서 사용할 변수를 초기화**합니다. **조건식은 루프 안쪽 코드가 실행되기 전에 실행**되며, **참이 아니면 루프를 종료**합니다. 이 식은 **초기화 직후에도 평가**될 수 있습니다. **증감문은 루프 안쪽 코드가 실행된 후에 실행**되는 문장으로, 보통 **루프 변수를 증감**시키는 용도로 쓰입니다.

*goto*는 여러 가지 용도로 사용되었지만, *for*는 오로지 루프만을 위해 사용되므로 가독성도 좋고, 컴파일러가 최적화해 줄 여지도 많습니다. 또, (정상적으로 사용한다는 가정 하에) *for*가 있는 한 줄만 봐도 이 루프가 얼마나 돌지 알 수 있기 때문에 여러모로 *goto*보다 좋습니다.

```
#include <stdio.h>
int main() {
    int k, i;
    scanf("%d", &k);
    for(i = 0; i < k; ++i) {
        printf("i: %d\n", i);
```

```
    }
    return 0;
}
```

```
4
```

```
i: 0
i: 1

i: 2
i: 3
```

여기서의 루프 변수는 *i*입니다. 관습적으로, 루프 변수는 *i*나 *j*같은 것을 많이 사용합니다.

*goto*와 레이블을 사용하면 이런 내용이겠지요.

```
#include <stdio.h>
int main() {
    int k, i;
    scanf("%d", &k);
    i = 0;
loop_start:
    if(!(i < k))
        goto loop_end;
    printf("i: %d\n", i);
```

```
    ++i;
    goto loop_start;
loop_end:
    return 0;
}
```

for 문을 사용한 쪽이 훨씬 낫지 않습니까?

*if*와 마찬가지로, *한 줄 뿐이라면 중괄호를 생략*할 수 있습니다. 이는 뒤의 *while*과 *do ~ while*에도 해당되는 내용입니다.

```
#include <stdio.h>
int main() {
    int k, i;
    scanf("%d", &k);
    for(i = 0; i < k; ++i)
        printf("i: %d\n", i);
    return 0;
}
```

for 문과 함께 변수를 선언할 수도 있습니다.

```
#include <stdio.h>
int main() {
    int k;
    scanf("%d", &k);
    for(int i = 0; i < k; ++i)
        printf("i: %d\n", i);
```

```
    return 0;
}
```

다만, 이때의 *i*는 *for* 문 안에서만 사용할 수 있습니다.

콤마를 사용해 여러 개의 초기화와 증감문 사용도 가능하고,

```
#include <stdio.h>
int main() {
    int k;
    scanf("%d", &k);
    for(int i = 0, j = 1; i < k; ++i, ++j)
        printf("i: %d, j: %d\n", i, j);
    return 0;
}
```

> 4

> i: 0, j: 1
> i: 1, j: 2
>
> i: 2, j: 3
> i: 3, j: 4

초기화, 조건식, 증감문 중 어떤 것을 비워 놔도 그냥 없는 셈 치고 돌아갑니다.

```
#include <stdio.h>
int main() {
	int k;
	scanf("%d", &k);
	for(int i = 0; i < k; )   //무한 루프입니다!
		printf("i: %d\n", i);
	return 0;
}
```

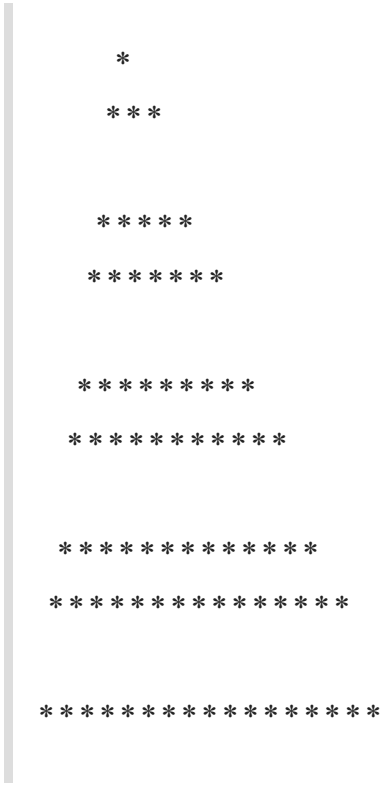
```
4
```

```
i: 0
i: 0
i: 0
i: 0
....
```

for 안에 *for*를 쓰고 싶다는 생각은 안해보셨나요?

```
#include <stdio.h>
int main() {
	for(int i = 1; i < 10; ++i) {
		for(int j = i; j < 9; ++j)
```

```
            putchar(' ');
        for(int j = 0; j < (i * 2 - 1); ++j)
            putchar('*');
        putchar('\n');
    }
    return 0;
}
```



고정폭 폰트로 보면 완전히 대칭인 모습을 볼 수 있습니다.

_for_문에서 _continue_를 만나면 그 _for_문의 시작으로 가 증감문을 실행하고 조건식으로 평가합니다. **_break_를 만나면 현재의 _for_문을 탈출**합니다.

```
#include <stdio.h>
int main() {
    for(int i = 0; i < 10; ++i) {
        if(i == 3) continue;
        printf("i: %d\n", i);
        if(i == 7) break;
    }
    return 0;
}
```

```
i: 0
i: 1

i: 2
i: 4

i: 5
i: 6

i: 7
```

*3*은 *continue* 때문에, *7* 이후는 *break* 때문에 더 이상 실행되지 않습니다. 다만, 여러 겹의 *for*로부터 탈출하려면 얄짤없이 *goto*를 사용해야 합니다.

for 문을 사용하면 배열의 원소들을 쉽게 넣고 꺼내 쓸 수 있습니다.

```
#include 〈stdio.h〉
int main() {
	int arr[ 5 ] = { 0, };
	for(int i = 0; i 〈 5; ++i)
		arr[ i ] = 5 - i;
	for(int i = 0; i 〈 5; ++i)
		printf("arr[ %d ]: %d\n", i, arr[ i ]);
	return 0;
}
```

```
arr[ 0 ]: 5
arr[ 1 ]: 4

arr[ 2 ]: 3
arr[ 3 ]: 2

arr[ 4 ]: 1
```

그리고 잘 보면 *i++*보다는 *++i*를 많이 사용하는 모습을 볼 수 있는데, 이는 ***++i***가 그냥 ***1*만큼 증가**시키는 반면 ***i++***은 ***i*를 복사한 후 원본 *i*가**

***1*만큼 증가**하는 방식으로 움직이기 때문입니다. 즉, ***++i*가 빠릅니다.** 물론 컴파일러에 따라 자동으로 *++i*로 바꿔주기도 하지만, 혹시 모르니 *++i*로 사용하는 것이 좋습니다.

while과 do ~ while

for 말고도 반복문을 만들 수 있는 수단이 존재하는데요, 바로 *while*과 *do ~ while*입니다.

***while(조건식) { 내용 }*의 형식으로 *while*문을 만들 수 있으며**, 이는 **조건식이 참인 동안 괄호 안의 내용을 실행**합니다. **조건식은 처음 시작하자마자 평가**됩니다.

```
#include <stdio.h>
int main() {
	int i = 0;
	while(i < 5)
		printf("%d\n", i++);
	return 0;
}
```

```
0
1
2
3
4
```

***do { 내용 } while(조건식)*의 형식으로 *do ~ while* 문을 만들 수 있으며**, 이는 **조건식이 참인 동안 괄호 안의 내용을 실행**합니다. *while*과의 차이점은 **일단 한 바퀴 실행**된다는 것입니다.

```
#include <stdio.h>
int main() {
	int i = 0;
	do
		printf("%d\n", i++);
	while(i < 5);
	return 0;
}
```

```
0
1
2
3
4
```

이 코드에서는 같은 역할을 하는 것처럼 보입니다. 다른 코드는 어떨까요?

```
#include <stdio.h>
int main() {
	int i = 0;
	while(i)
		printf("퉤에에엣\n");
	return 0;
}
```

```
#include <stdio.h>
int main() {
	int i = 0;
	do
		printf("퉤에에엣\n");
	while(i);
	return 0;
}
```

> 퉤에에엣

첫 번째 코드에서는 아무 것도 실행되지 않지만, 두 번째 코드에서는 *퉤에에엣*이 출력되는 모습을 볼 수 있습니다. *goto*와 레이블을 사용한 코드로 바꾸어서 생각해 보면 더 쉽게 이해할 수 있을 것입니다.

switch ~ case

*if(조건식) { 내용 } else if(조건식) { 내용 } else if(조건식) { 내용 }...*을 반복하는 코드는 가독성이 떨어집니다. 이를 해결하기 위해 ***switch ~ case***문이 존재합니다.

```
#include <stdio.h>
int main() {
    int a;
    scanf("%d", &a);
    switch(a) {
    case 0:
        printf("Zero\n");
        break;
    case 1:
        printf("One\n");
        break;
    case 2:
        printf("Two\n");
        break;
    case 3:
        printf("Three\n");
        break;
    default:
        printf("Other\n");
        break;
    }
```

```
    return 0;
}
```

> 0
>
> Zero

> 1
>
> One

> 2
>
> Two

> 3
>
> Three

> 100
>
> Other

***switch ~ case*문에서는 *switch*에 들어가는 수에 해당하는 라벨로 점프**합니다. **해당하는 라벨이 없다면 *default* 라벨로 점프**합니다. *switch ~ case*문은 *if*문과 달리 점프할 뿐이므로 **아래의 코드들을 실행하고 싶지 않다면 *break*를 통해 *switch ~ case*문을 멈추어야 합니다.**

```
#include <stdio.h>
int main() {
    int a;
    scanf("%d", &a);
    switch(a) {
    case 0:
        printf("Zero\n");
    case 1:
        printf("One\n");
    case 2:
        printf("Two\n");
    case 3:
        printf("Three\n");
    default:
        printf("Other\n");
    }
    return 0;
}
```

> 1

> One
> Two
>
> Three
> Other

***switch*에 들어가는 값은 항상 정수**여야 합니다. 또, **레이블의 수는 반드시 정수형 컴파일 타임 상수**여야 합니다. **const는 런타임 상수**입니다! 정수만 가능한 것은 아까 잠시 보여주었던 레이블의 배열과 관련이 있습니다. 첫 번째 *switch ~ case*문을 *goto*와 *if*로 바꾸면(컴파일 결과로 나온 어셈블리어를 있는 그대로 C 언어로 옮기면)

```
#include <stdio.h>
int main() {
    int a;
    scanf("%d", &a);
    if(a > 3)
        goto other;
    if(a == 0)
        goto l0;
    if(a == 1)
        goto l1;
    if(a == 2)
        goto l2;
    if(a == 3)
        goto l3;
l0:
    printf("Zero\n");
    goto switch_end;
l1:
    printf("One\n");
    goto switch_end;
l2:
```

```
    printf("Two\n");
    goto switch_end;
l3:
    printf("Three\n");
    goto switch_end;
other:
    printf("Other\n");
    goto switch_end;
switch_end:
    return 0;
}
```

이런 코드가 나옵니다. 조금 더 똑똑한 컴파일러는

```
#include <stdio.h>
int main() {
    int a;
    scanf("%d", &a);
    void* lable[ 4 ] = { &&l0, &&l1, &&l2, &&l3 };
    if(a > 3)
        goto other;
    goto *lable[ a ];
l0:
    printf("Zero\n");
    goto switch_end;
l1:
    printf("One\n");
    goto switch_end;
```

```
l2:
    printf("Two\n");
    goto switch_end;
l3:
    printf("Three\n");
    goto switch_end;
other:
    printf("Other\n");
    goto switch_end;
switch_end:
    return 0;
}
```

이렇게 바꿔주는데, 대부분의 경우 배열의 *n*번째 원소에 접근하는 것은 무척 싼 작업이나 *if* 문을 사용하는 것은 비싼 작업이기 때문에 후자가 **성능을 높이는 데 유리**합니다. 이러한 작업을 위해 C 언어에서는 *switch ~ case* 문에서 **정수만 사용**할 수 있게 합니다. 물론 언어나 컴파일러에 따라서는 그렇지 않은 경우도 있습니다.

컴퓨터와 놀기

함수

함수는 **특정 동작을 수행하는 일정 코드 부분**입니다. ***반환하는 값의 자료형 함수명 (인자)***의 형식으로 선언하며(반환 값의 자료형의 ***void*는 아무 값도 반환하지 않는다**는 뜻입니다), 어떤 함수를 호출하기 위해서는 **그 함수를 호출하는 부분 위에 그 함수가 정의**되어 있어야 합니다. 반환하는 값의 자료형이 *int*라면 생략할 수 있으며. 인자도 없다면 괄호 안을 생략할 수 있습니다. 반환하는 값의 자료형 앞에 ***inline*을 붙여 주면 함수를 호출하는 대신 호출한 곳의 내용이 함수의 내용으로 치환**됩니다. 이 과정은 컴파일러에 의해 수행되며, 경우에 따라서는 프로그램의 수행 속도를 빠르게 해 줄 수도 있습니다. 하지만 그렇지 않은 경우도 있으니 유의해서 사용하십시오,

```
#include <stdio.h>
void func() {
    printf("퉤에에엣\n");
}
int main() {
    for(int i = 0; i < 4; ++i)
    func();
    return 0;
}
```

> 퉤에에엣

> 퉤에에엣
>
> 퉤에에엣
>
> 퉤에에엣

서로 다른 두 함수는 변수를 따로 사용합니다. 예를 들어, *main* 함수에서 *i*를 선언했어도, *asdf* 함수에서는 그 i를 사용하지 못합니다. 만일 *asdf* 함수에서 새로 *i*를 선언한다면, 그것은 *main*의 *i*와는 다른 *i*입니다.

```
#include 〈stdio.h〉
void func() {
	//printf("%d\n", i);	i가 선언되지 않았습니다!
	for(int i = 0; i 〈 2; ++i)
		printf("퉤에에엣 ");
	printf("\b\n");
}
int main() {
	for(int i = 0; i 〈 4; ++i)
	func();
	return 0;
}
```

> 퉤에에엣 퉤에에엣
> 퉤에에엣 퉤에에엣
>
> 퉤에에엣 퉤에에엣

> 퉤에에엣 퉤에에엣

***반환하는 값의 자료형 함수명 (인자)*의 형식**으로 **함수 원형**을 적어주면 **함수의 정의부가 호출하는 곳보다 아래에 있어도 컴파일이 가능**합니다. 단, **함수 원형은 함수 호출보다 위**에 있어야 합니다.

```
#include <stdio.h>
void func();
int main() {
	for(int i = 0; i < 4; ++i)
	func();
	return 0;
}
void func() {
	printf("퉤에에엣\n");
}
```

> 퉤에에엣
>
> 퉤에에엣
>
> 퉤에에엣
>
> 퉤에에엣

인자는 변수를 선언하듯 ***자료형 이름***으로 써줍니다. **자료형이 같아도 모든 인자 이름 앞에 자료형을 적어주어야 합니다.**

```
#include <stdio.h>
void print(int a) {                //정수 인자를 1개 받으며, 그 수는 print 함수의
    printf("a: %d\n", a);          //a 라는 int 변수 속에 들어가 있습니다.
}
int main() {
    for(int i = 0; i < 4; ++i)
        print(i);
    return 0;
}
```

```
a: 0
a: 1

a: 2
a: 3
```

print 함수의 *a*는 *main* 함수의 *i*와 완벽히 동떨어진 존재입니다. 값을 공유하고 싶다면 변수를 함수 밖에 선언하거나(**전역 변수**), 포인터를 넘겨줍니다★. 다만 캐시 적중률이나 후술할 안전성 등의 이유 때문에 **전역 변수 사용은 별로 권장되지 않습니다.**

```
#include <stdio.h>
void func(int a) {
	++a;
}
int main() {
	int k = 4;
	func(k);
	printf("k: %d\n", k);
	return 0;
}
```

> k: 4

정 수정하고 싶다면 포인터를 넘겨줍니다★.

```
#include <stdio.h>
void func(int* a) {
	++(*a);
}
int main() {
	int k = 4;
	func(&k);
	printf("k: %d\n", k);
	return 0;
}
```

> k: 5

```
#include <stdio.h>
void func() {
    int a = 0;
    static int b = 0;
    ++a;
    ++b;
    printf("a: %d, b: %d\n", a, b);
}
int main() {
    for(int i = 0; i < 3; ++i)
        func();
    return 0;
}
```

```
a: 1, b: 1
a: 1, b: 2

a: 1, b: 3
```

함수는 호출될 때마다 변수를 초기화합니다. 하지만 **자료형 앞에 *static* 을 붙히면** 함수가 몇 번이고 호출되어도 **변수가 초기화되지 않습니다.**

```
#include <stdio.h>
int addthree(int a) {
    return a + 3;
}
```

```
int main() {
	for(int i = 0; i < 4; ++i)
		printf("addthree(%d): %d\n", i, addthree(i));
	return 0;
}
```

```
addthree(0): 3
addthree(1): 4

addthree(2): 5
addthree(3): 6
```

C에서의 함수는 반환값이 있다면 **인자와 반환값의 대응관계**라 할 수 있습니다. **함수가 반환하는 값은 *return* 뒤**에 적어주며, **함수는 *return*을 만나면 그 자리에서 즉시 값을 반환하고 종료**됩니다. 눈치가 빠른 사람이라면 우리가 지금까지 썼던 ***main*도 사실 함수**이며, 다른 함수와의 차이점은 프로그램의 시작과 함께 가장 먼저 시작된다는 것뿐이라는 것을 알았을지도 모릅니다.

C에서의 함수는 수학에서의 함수와 약간 의미가 다를 수 있습니다. 우선, 값을 반환하지 않는 함수가 존재합니다. 또, 수학에서의 함수는 오로지 인자와 결과값의 대응 관계로, 인자가 같다면 결과값도 언제나 같지만, C에서는 ***static* 변수나 전역 변수의 존재로 인해 인자가 같을 때 결과값도 언제나 같다는 보장을 할 수 없습니다**. 이 때, 인자가 같을 때 결과값이 언제나 같다는 성질을 **참조 투명성**이라 하며, 이러한 성질을

갖는 함수를 **순수한 함수**라 합니다. 또, 함수의 실행이 외부에 영향을 미치는 상황(전역 변수 수정 등)을 **사이드 이펙트**라고 부릅니다.

함수가 사이드 이펙트를 일으키지 않는다면, 이 함수들은 동시 처리(**병렬 처리**)가 가능합니다. 여기에서는 다루지 않았지만, **멀티 스레딩**을 통해 서로 상관관계가 없는 함수들을 동시에 실행시킬 수 있습니다.

다른 함수로 *goto*를 사용해 **점프하는 것은 불가능**합니다. 아까 보여주었던, 레이블을 변수처럼 다루는 것을 사용해도, 프로그램은 문제를 일으키며 종료될 것입니다. 굳이 다른 함수 중간으로 건너뛸 일이 있다면, *setjmp/longjmp*를 사용하세요.

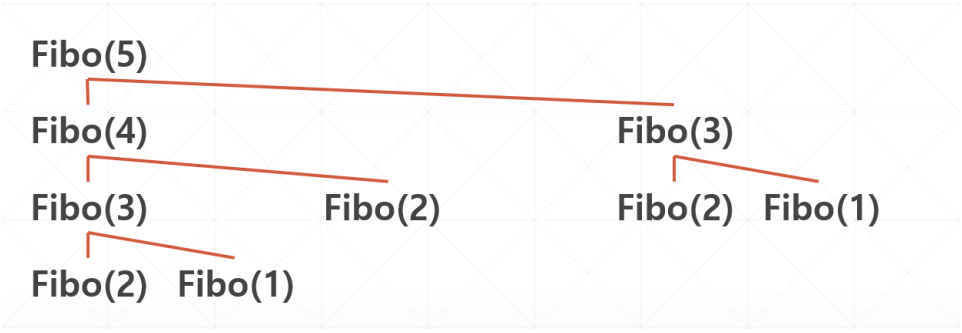
재귀 함수

재귀 함수는 자기 자신을 호출하는 함수입니다. 점화식과 비슷한데, 예시를 보면 좀 더 잘 이해할 수 있습니다.

```
#include <stdio.h>
int fibo(int a) {
    return (a < 3) ? 1 : fibo(a - 1) + fibo(a - 2);
}
int main() {
    for(int i = 1; i < 7; ++i)
        printf("fibo(%d): %d\n", i, fibo(i));
    return 0;
}
```

> fibo(1): 1
> fibo(2): 1
>
> fibo(3): 2
> fibo(4): 3
>
> fibo(5): 5
> fibo(6): 8

이름에서 눈치챘을 수도 있겠지만, 이 함수는 피보나치 수열의 *a*항을 구해주는 함수입니다. *a*가 *3* 미만, 즉 *1*과 *2*이라면 *1*을 반환하고, 그렇지 않다면 전 항과 전의 전 항을 반환합니다. 그런데, 이 피보나치 수열 함수는 조금 느리게 움직입니다.



잘 보면, *fibo(a)*를 구하기 위해 호출하는 함수의 갯수가 *fibo(a)*에 비례해서 커집니다. *fibo(a)*는 1.618^a에 비례해 커지므로, 결론적으로 *fibo(a)*를 구하기 위해 걸리는 시간은 1.618^a에 비례해 커집니다. 조금 더 개선할 수 있을 것 같습니다.

```
int fibo(int a) {
    static int size = 3, arr[ 100 ] = { 0, 1, 1, 0, };
    if(a >= size) {
        arr[ a ] = fibo(a - 1) + fibo(a - 2);
        ++size;
    }
    return arr[ a ];
}
```

자, 이렇게 하면 배열에 이미 구해둔 값들을 저장해 놓으니까 좀 낫죠? 이미 구한 값은 바로바로 꺼내서 쓸 수 있고. 이런 것을 **메모이제이션**이라 합니다. 여기서는 걸리는 시간이 a에 비례해서 커지겠군요. 그런데 여기서 문제가 생겼습니다. 수가 많아지다 보니, *arr[100]*으로는 모자라는 경우가 생겼습니다. *200*으로 늘렸지만 한참 모자랍니다(사실 *94*항만 해도 *unsigned long long*의 최대 크기인 $2^{64} - 1$을 넘어갑니다). 어떻게 해야 할까요?

```
#include <stdio.h>
int fibo(int n, int a, int b) {
    return (n == 1) ? b : fibo(n - 1, b, a + b);
}
int main() {
    for(int i = 1; i < 7; ++i)
        printf("fibo(%d): %d\n", i, fibo(i, 0, 1));
    return 0;
}
```

```
fibo(1): 1
fibo(2): 1

fibo(3): 2
fibo(4): 3

fibo(5): 5
fibo(6): 8
```

일반적으로 같은 일을 하는 **재귀 함수와 반복문 중 속도가 더 빠른 것은 반복문**입니다. 함수 호출에 따른 오버헤드가 없기 때문입니다. 하지만 컴파일러는 ***return* 뒤에 수식(연산자) 없는 값이 오면 꼬리 재귀 최적화**라고 불리는 것을 해 주는데, 이것을 통해 재귀 함수는 반복문과 동등한 수준의 속도를 얻게 됩니다(물론 반복문 쪽이 약간 더 빠릅니다). 이 코드는 꼬리 재귀를 사용할 뿐만 아니라, *a*와 *b* 두 변수에만 전 항과 전의 전 항을 저장해 메모리를 아낍니다. 이런 기법을 **슬라이싱**이라고 합니다. 하지만 두 번째 코드는 *fibo(10)*을 구한 후 추가적인 함수 호출 없이 *fibo(5)*를 구할 수 있었지만, 이번 코드는 얄짤 없이 처음부터 계산해야 합니다. 장단점이 있지요.

특정 작업들은 재귀함수를 사용하면 아주 쉽게 해결이 가능한 경우가 있습니다. 하지만 **재귀 함수를 사용하는 알고리즘은 일반적으로 느리고**(같은 부분을 여러번 계산하기 때문), 또 설령 같은 알고리즘으로 동작한다 해도 함수 호출에 의한 오버헤드가 발생합니다. 꼬리 재귀 최적화도 아무 컴파일러나 해 주는 것이 아닙니다. **반복문으로도 해결 가능한 작업이라면, 반복문을 사용해 주세요**.

조금 더 머리를 굴리면 *log a*에 비례한 시간만으로도 *fibo(a)*를 구할 수 있습니다(사실은 $log_2 a$지만, 밑 변환을 해 주면 상수배입니다). 또, C++이라는 언어에서는 *a*가 *1*이던 *100*이던 같은 시간 안에 *fibo(a)*를 출력할 수 있습니다(물론 여기서 말하는 시간은 컴파일하는 데 걸리는 시간을 제외한 것입니다). 잘 고민해 보세요!

포인터 1

포인터는 C언어의 꽃입니다. 잘 사용하면 아주 좋지만 C 언어로 작성된 프로그램의 버그의 90%는 포인터에서 나온다고 할 수 있을 정도로 어렵고, 또 위험한 개념입니다. 이것을 사용하면 컴퓨터를 무척 세심하게 조절할 수 있지만, 앞에서 말한 위험성 때문에 최근의 고급 언어들에서는 대부분 포인터를 사용하지 않습니다. 그러나 C 언어는 컴퓨터 운영체제를 작성하기 위해 제작된 언어이므로 포인터가 반드시 필요합니다. 읽기 어려울 수도 있지만 잘 읽고 사용할 수 있길 바랍니다. 컴퓨터 구조에 관련되어 있어 조금 어려운 부분도 있는데, 그런 부분은 ☆표를 해 놓았으니 너무 어렵다면 넘어가세요.

우리가 **변수를 선언하면**, 컴퓨터 **메모리상의 어딘가에 그 변수의 값을 담아두게 됩니다**. 예를 들어 *int a = 5;*를 하게 되면 *int*는 32bit 컴퓨터에서 4바이트 정수 자료형이므로 메모리상에 *a*를 위한 4바이트의 공간이 *a*를 위해 할당되고 그 안에 *5*라는 값이 들어갑니다. 또, 우리는 *&*연산자를 사용해 *a*의 메모리 주소를 알 수 있습니다. 비트 AND 연산자 *&*와는 다릅니다(전자는 *&a*처럼 사용하는 단항 연산자지만 후자는 a & b처럼 사용하는 이항 연산자입니다). 또, *연산자를 사용하여 어떤 주소로부터 주소가 가리키는 값을 가져올 수 있습니다. *int a = 5;*라면 **&a*는 *5*입니다. 어떤 자료형의 주소를 담는 자료형은 *자료형**로 사용합니다. 주소의 주소는 ****, 주소의 주소의 주소는 *****과 같은 식입니다. 또, 이런 주소 자료형의 크기는 32bit에서 4바이트, 64bit에서 8바이트입니다. 다음의 예시를 보겠습니다.

```
#include <stdio.h>
int main() {
	char arr[ 4 ] = { 0, 1, 2, 3 };
	int a = 5;
	int* b = &a;
	for(int i = 0; i < 4; ++i)
		printf("arr[ %d ]: %d...%X\n", i, arr[ i ], &arr[ i ]);
	printf("a: %d...%X\n", a, &a);
	printf("b: %X...%X...%X\n", *b, b, &b);
	return 0;
}
```

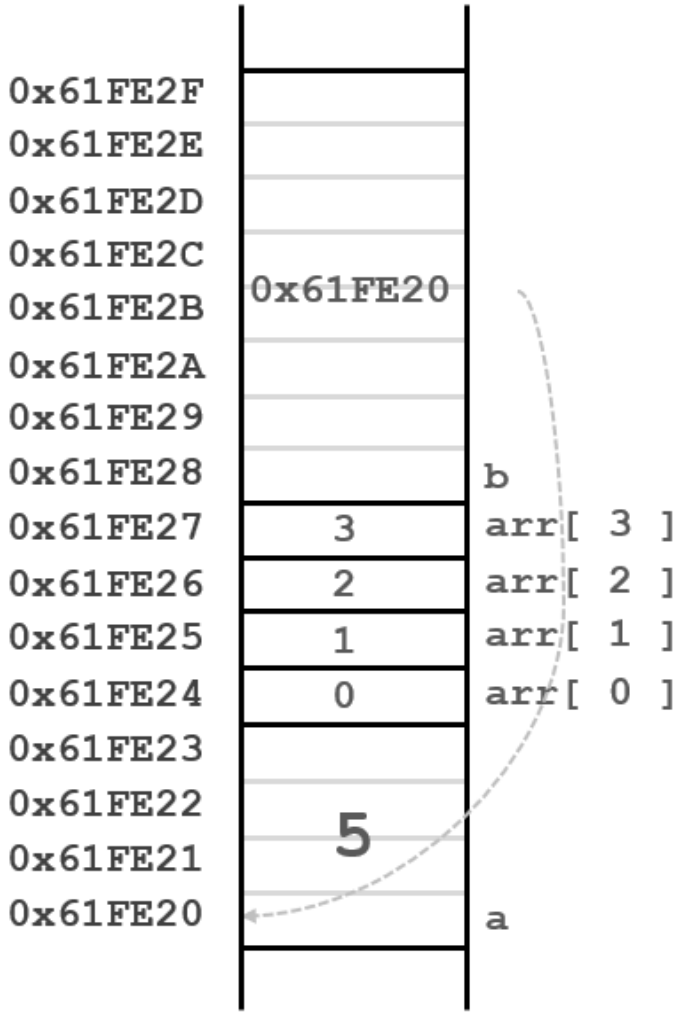
```
arr[ 0 ]: 0...61FE24
arr[ 1 ]: 1...61FE25

arr[ 2 ]: 2...61FE26
arr[ 3 ]: 3...61FE27

a: 5...61FE20
b: 5...61FE20...61FE28
```

모든 코드는 *int*가 4바이트, 주소값이 8바이트라는 전제 하에 작성되었으며, 주소값은 실행 환경에 따라 달라질 수 있다는 걸 기억하세요.

이 코드에서 각 변수가 어떻게 저장되어 있을까요?



4바이트 정수형인 *a*는 *0x61FE20*부터 4바이트를 차지하고 있고, 배열 *arr*는 1바이트 정수의 배열이기 때문에 각 원소가 1바이트씩, *0x61FE24*부터 총 4바이트를 차지하고 있습니다. *a*에게 주어진 4바이트의 공간에는 *5*가, *arr*의 원소들에게 주어진 1바이트의 공간에는 각각 *1, 2, 3, 4*가 들어있네요.

주목해서 보아야 할 것은 *b*입니다. *b*는 *int**이므로 8바이트의 크기를 갖습니다. 또한, 이 8바이트에는 *0x61FE20*이 저장되어 있습니다. 우리가 *b*에 *&a*, 즉 *a*의 주소를 담았기 때문에 그렇습니다. 따라서 그냥 *b*는 *0x61FE20*, *b*값(*0x61FE20*)이 가리키는 곳의 값은 *5(a)*, *b*의 주소는 *0x61FE28*입니다.

참고로 **리틀 엔디안** 방식의 컴퓨터에서는 **하위 비트**(낮은 자릿수)**가 낮은 주소값** 쪽에, **상위 비트**(높은 자릿수)가 **높은 주소값** 쪽에 저장되어 있으며, **빅 엔디안 방식은 그 반대**입니다☆.

한 함수 안에서 각 변수들 중 어느 것이 높은 주소에 있을지, 어느 것이 낮은 주소에 있을지는 아무도 알 수 없습니다. 컴파일러가 임의로 자리를 바꿔버리기 때문입니다. 레지스터의 크기에 맞게 변수를 정렬aligned하면 더 빠르게 동작하기 때문에, 컴파일러는 변수의 위치를 막 바꿀 수도 있습니다☆. 예를 들어 SIMD 명령어 *movdqa*를 사용하려면 16바이트로 정렬되어야 하지요☆. **'이 녀석들은 연속되어 있다!'라고 확실하게 말할 수 있는 것은 배열의 원소뿐**입니다. 그렇다면 다른 함수에서 선언된 변수들이나 전역변수는 어떨까요?

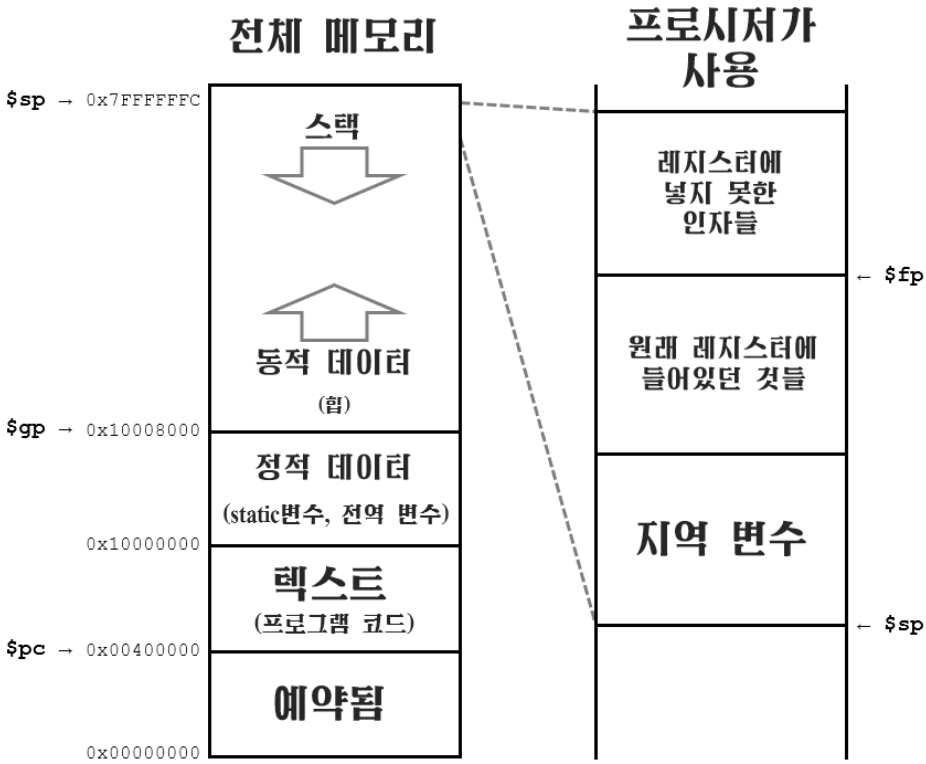
```
#include <stdio.h>
int e = 2, f = 1;
void func() {
    char c = 4;
    static char d = 3;
    printf("c: %d...%X\n", c, &c);
    printf("d: %d...%X\n", d, &d);
}
```

```
int main() {
	char a = 6, b = 5;
	printf("a: %d...%X\n", a, &a);
	printf("b: %d...%X\n", b, &b);
	func();
	printf("e: %d...%X\n", e, &e);
	printf("f: %d...%X\n", f, &f);
	return 0;
}
```

```
a: 6...61FE4E
b: 5...61FE4F

c: 4...61FE0F
d: 3...403010

e: 2...403018
f: 1...403014
```



함수에서 선언한 변수(**지역 변수**)들은 컴퓨터의 **스택** 영역에 자리하고, ***static*으로 선언한 변수나 전역 변수**는 **정적 데이터** 영역에 자리합니다. 여기서는 다루지 않지만, **동적 할당한 변수**들은 **동적 데이터**(**힙**) 영역에 자리합니다☆. 여기 나와 있는 주소는 관례일 뿐, 실제로 그렇다는 보장은 없습니다. 또, 프레임 포인터(*$fp*)의 경우, 이것을 사용하지 않는 컴파일러도 있습니다☆. GCC에서는 사용한다.

변수들의 주소를 보면, *c*가 있는 곳은 *a*와 *b*가 있는 곳으로부터 좀 떨어져 있습니다. 둘 다 스택 영역에 있는데도 말이죠. 이 공간에는 레지스

터에 넣지 못한 인자들(여기에서는 없습니다)이나 원래 레지스터에 들어있던 것들이 저장되어 있습니다☆. 따라서 **함부로 이 데이터들을 수정했다가는 프로그램이 오류를 내며 종료**될 것입니다.

주소 자료형을 사용할 때 주의해야 할 점이 있습니다.

```
int main() {
	int* a, b;		//a는 int*, b는 int
	int* c, * d;	//c는 int*, d도 int
	return 0;
}
```

typedef 원래 자료형 별칭을 사용해 한 자료형의 **별칭**을 만들 수 있는데,

```
#include <stdio.h>
typedef int jungsoo;
int main() {
	jungsoo a, b;	//a와 b는 jungsoo
	printf("%d %d\n", sizeof a, sizeof b);
	return 0;
}
```

> 4 4

이것을 사용하면 문제를 피해갈 수 있습니다.

```
#include <stdio.h>
typedef int* int_p;
int main() {
	int_p a, b; //a와 b는 int*
	printf("%d %d\n", sizeof a, sizeof b);
	return 0;
}
```

> 8 8

하지만 되려 코드의 가독성을 해칠 수 있으니 '이런 것이 있다' 정도만 알아두세요.

주소 자료형의 연산은 조금 독특합니다. 일반 정수 자료형의 덧셈과 뺄셈에서는 *+ 1*을 하면 값이 *1* 증가하고 *- 1*을 하면 값이 *1* 감소했지만 주소 자료형은 다릅니다. **주소 자료형 *T**에서 *+ 1*을 하면 값이 *sizeof(T)*만큼 증가**하고, **- *1*을 하면 *sizeof(T)*만큼 감소**합니다. 예시를 볼까요?

```
#include <stdio.h>
int main() {
	int a = 5;
	int* b = &a;
	printf("int의 크기는 %d\n", sizeof(int));
	printf("b - 2: %X\n", b - 2);
	printf("b - 1: %X\n", b - 1);
	printf("b    : %X\n", b);
	printf("b + 1: %X\n", b + 1);
```

```
    printf("b + 2: %X\n", b + 2);
    return 0;
}
```

```
int의 크기는 4
b - 2: 61FE44

b - 1: 61FE48
b      : 61FE4C

b + 1: 61FE50
b + 2: 61FE54
```

주소 자료형 중에는 조금 독특한 것이 있는데, 바로 ***void****입니다. '없는 것을 가리키다니! 이게 무슨 짓이야!'라고 생각할 수도 있지만, ***void****은 모든 타입을 가리킬 때 사용*할 수 있으며, **덧셈과 뺄셈을 하면 1씩 증가하고 감소**합니다. 나중에 가변 인자에 대해 공부할 때 보게 될 것입니다.

지금까지 꽁꽁 숨겨왔던 비밀 그 첫 번째, 배열과 포인터의 관계를 공개합니다!

```
#include <stdio.h>
int main() {
    int arr[ 5 ] = { 0, };
```

```
    printf("&arr[ 0 ]: %X\n", &arr[ 0 ]);
    printf(" arr    : %X\n", arr);
    return 0;
}
```

```
&arr[ 0 ]: 61FE30
 arr       : 61FE30
```

배열의 이름은 사실 **그 배열 첫 번째 원소의 주소**를 의미합니다. 저번에 연산자를 공부하며 지나가는 말로 ***a[b]*와 **(a + b)*가 같은 뜻**이라고 한 것을 기억하나요? 이 이유로 배열의 인덱스는 *1*이 아니라 *0*부터 시작합니다. 또, arr*[0]* 뿐만 아니라 *0[arr]* 또한 유효한 코드입니다.

***const*를 자료형 앞에 붙이면 그 변수가 가리키는 주소의 값을 변경할 수 없고, 뒤에 붙이면 그 변수가 가리키는 주소를 변경할 수 없습니다.**

```
int main() {
    int a = 5, b = 10;
    const int* ptr1 = &a;
    int* const ptr2 = &a;
    //*ptr1 = 10;              불가능
    ptr1 = &b;
    *ptr2 = 10;
    //ptr2 = &b;              불가능
    return 0;
}
```

다차원 배열의 원소들이 사실 메모리 상에서는 일직선상에 있다고 했던 것도 기억합니까?

```
#include <stdio.h>
int main() {
	int arr[ 2 ][ 2 ] = { 0, };
	printf("&arr[ 0 ][ 2 ]: %X\n", &arr[ 0 ][ 2 ]);
	printf("&arr[ 1 ][ 0 ]: %X\n", &arr[ 1 ][ 0 ]);
	return 0;
}
```

> &arr[0][2]: 61FE48
> &arr[1][0]: 61FE48

아, 그리고 우리는 '배운 사람'이므로 위의 코드를 조금만 고쳐줍시다.

```
#include <stdio.h>
int main() {
	int arr[ 2 ][ 2 ] = { 0, };
	printf("&arr[ 0 ][ 2 ]: %X\n", arr[ 0 ] + 2);
	printf("&arr[ 1 ][ 0 ]: %X\n", arr[ 1 ]);
	return 0;
}
```

여러분은 포인터를 떡 주무르듯 다룰 수 있어야 합니다. 연습! 또 연습!

```
#include 〈stdio.h〉
int main() {
      int a[ 3 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
      for(int i = 0; i 〈 9; ++i)
            printf("%X %d  %X %d  %X %d\n",
            *a + i,
           *(*a + i),
            (void*)a + sizeof(int) * i,
    *(int*)((void*)a + sizeof(int) * i),
            (void*)a + sizeof(int[ 3 ]) * (i / 3) + sizeof(int) * (i % 3),
    *(int*)((void*)a + sizeof(int[ 3 ]) * (i / 3) + sizeof(int) * (i % 3)));
      return 0;
}
```

```
61FE00 1 61FE00 1 61FE00 1
61FE04 2 61FE04 2 61FE04 2

61FE08 3 61FE08 3 61FE08 3
61FE0C 4 61FE0C 4 61FE0C 4

61FE10 5 61FE10 5 61FE10 5
61FE14 6 61FE14 6 61FE14 6

61FE18 7 61FE18 7 61FE18 7
61FE1C 8 61FE1C 8 61FE1C 8
```

> 61FE20 9 61FE20 9 61FE20 9

```
#include <stdio.h>
int main() {
	char arr[ 4 ] = { 0, };
	*(int*)arr = 0x78563412;
	for(int i = 0; i < 4; ++i)
		printf("arr[ %d ]: %X...%X\n", i, arr[ i ], arr + i);
	return 0;
}
```

> arr[0]: 12...61FE3C
> arr[1]: 34...61FE3D
>
> arr[2]: 56...61FE3E
> arr[3]: 78...61FE3F

두 번째 코드는 리틀엔디언Little-endian 방식 컴퓨터가 아니라면 *12 34 56 78*이 나오지 않을 수 있습니다☆.

자, 약속을 지킬 시간입니다. 큰 따옴표로 둘러싸인 문자열의 비밀에 대해 많이들 궁금했을 것입니다.

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

```
    char* a = "BOY";
    printf("%X\n%X\n", a, "BOY");
    return 0;
}
```

```
404000
404000
```

여러분이 *"BOY"*를 만들면, **정적 데이터 영역에** *'B', 'O', 'Y', '₩0'*으로 구성된 **배열이 생깁니다.** 또, *"BOY"* 그 자체는 **그 배열 첫 번째 원소의 주소**입니다. 그러니까, 조금 장난을 쳐보자면, *2["BOY"]*와 같은 코드도 유효하며, 그 값은 *'Y'*입니다. 또다시 *"BOY"*를 사용할 일이 있으면, 아까 만든 *"BOY"*를 재사용할 수 있습니다. 하지만 이 **배열의 원소를 수정하지는 마십시오.** 프로그램이 오류를 일으키며 종료될 수 있습니다.

scanf 함수와 *printf* 함수가 문자열 출력을 위해 주소값만 받는다는 사실을 떠올리십시오. *%s*로 문자열을 출력할 때 여러분이 *printf*에 보냈던 것은 **배열 첫 원소의 주소**입니다. 만약 *arr*의 두 번째 원소부터 출력하고 싶다면 *printf("%s₩n", arr + 1)*과 같이 하면 되겠지요.

포인터 2

함수 부분에서 포인터를 넘겨 함수 밖의 원소를 수정한 것을 기억합니까? 여기에서는 함수에 인자로 포인터를 넘기는 것과 함수의 포인터를 다루도록 하겠습니다.

```
#include <stdio.h>
void func(int* a) {
    ++(*a);
}
int main() {
    int k = 4;
    func(&k);
    printf("k: %d\n", k);
    return 0;
}
```

> k: 5

아까 보았던 코드입니다. *func* 함수가 받고 있는 것은 *k*의 주소입니다. 물론 **'주소라는 값' 자체는 복사되지만, *k*는 복사되지 않습니다**. *func* 함수에서는 ***연산자를 통해 *a*가 가리키는 곳, 즉 *main* 함수의 *k*에 접근**할 수 있습니다.

같은 방법으로, **배열**을 함수에 넘겨주는 것도 가능합니다.

```
#include <stdio.h>
void func1(int* p) {
	printf("%X\n", p);
	for(int i = 0; i < 5; ++i)
		printf("%d ", p[ i ]);
	printf("\n%d\n", sizeof p);
}
void func2(int p[]) {
	printf("%X\n", p);
	for(int i = 0; i < 5; ++i)
		printf("%d ", p[ i ]);
	printf("\n%d\n", sizeof p);
}
int main() {
	int arr[ 5 ] = { 1, 2, 3, 4, 5 }, i;
	printf("%X\n", arr);
	for(int i = 0; i < 5; ++i)
		printf("%d ", arr[ i ]);
	printf("\n%d\n", sizeof arr);
	func1(arr);
	func2(arr);
	return 0;
}
```

```
61FE10
1 2 3 4 5
```

```
20
61FE10

1 2 3 4 5
8

61FE10
1 2 3 4 5

8
```

하지만 주의하세요, *func1*과 *func2*가 진짜로 받은 것은 ***arr* 배열이 아니라 단지 배열의 시작 주소**입니다.

자, 여기까지 읽었으면 지금껏 포인터에 관련된 부분이라며 건너뛰었던 내용들을 모두 이해할 수 있을 것입니다. 복습할 겸, 다시 처음으로 가서 ★이 있는 문장들을 주의 깊게 읽고, 이해하십시오.

함수를 가리키는 포인터를 만들 수도 있습니다. ***함수 반환값의 자료형 (*이름)(인자)***의 형식으로 만듭니다.

```
#include 〈stdio.h〉
void func1(int a, int b) {
	printf("a: %d, b: %d\n", a, b);
}
int func2(int a) {
	printf("퉤에에엣\n");
```

```
    return a + 1;
}
void func3() {
    printf("GIRL Is Recursion Lover\n");
}
void func4() {
    printf("BOY is Object-oriented programming Yearner\n");
}
int main() {
    void (*fp1)(int, int) = func1;
    int  (*fp2)(int)      = func2;
    void (*fparr[ 2 ])()  = { func3, func4 };
    fp1(2017, 2018);
    printf("%d\n", fp2(2018));
    fparr[ 0 ]();
    fparr[ 1 ]();
    return 0;
}
```

> a: 2017, b: 2018
> 튀에에엣
>
> 2019
> GIRL Is Recursion Lover
>
> BOY is Object-oriented programming Yearner

이것도 함수의 인자로 넘겨줄 수 있고,

```
#include <stdio.h>
void func1(int p) {
    printf("%d\n", p);
}
void (*func2(int a, int b))(int) {
    printf("%d %d\n", a, b);
    return func1;
}
int main() {
    func2(1, 2)(3);
    return 0;
}
```

```
1 2
3
```

함수의 반환값으로 사용할 수도 있습니다. 아, 너무 복잡하니 *typedef* 로 줄여달라고요?

```
#include <stdio.h>
typedef void (*func_p)(int);
void func1(int p) {
    printf("%d\n", p);
}
```

```
func_p func2(int a, int b) {
	printf("%d %d\n", a, b);
	return func1;
}
int main() {
	func_p (*fp)(int, int) = func2;
	func2(1, 2)(3);
	fp(4, 5)(6);
	return 0;
}
```

```
1 2
3
4 5
6
```

네, 해드렸습니다. 열심히 갖고 놀아보세요.

구조체

구조체는 기존의 자료형인 *int, char...*등을 묶어 만든 사용자 정의 자료형입니다. 구조체는 ***struct{ 내용 }***의 형식으로 만들 수 있습니다. 또, 구조체 자료형의 변수를 만들려면 ***struct 구조체 이름 변수 이름***과 같이 해 주면 됩니다.

```
#include 〈stdio.h〉
struct st1 {
    int a, b;
};
struct st1 p;
struct st2 {
    int c, d;   //멤버 이름이 겹쳐도 딱히 문제는 없습니다.
} q, k;          //바로 써도 OK
int main() {
    struct st1 ob1;
    struct st2 ob2;
    struct { int e, f; } ob3; //같은 자료형으로 더 이상 변수를 만들 생각이 없다면 OK
    p.a = 1, p.b = 2;
    q.c = 10, q.d = 20;
    ob1.a = 3, ob1.b = 4;
    ob2.c = 30, ob2.d = 40;
    ob3.e = 5, ob3.f = 6;
    printf("p.a: %d, p.b: %d\n", p.a, p.b);
```

```
    printf("q.c: %d, q.d: %d\n", q.c, q.d);
    printf("ob1.a: %d, ob1.b: %d\n", ob1.a, ob1.b);
    printf("ob2.c: %d, ob2.d: %d\n", ob2.c, ob2.d);
    printf("ob3.e: %d, ob3.f: %d\n", ob3.e, ob3.f);
    return 0;
}
```

```
p.a: 1, p.b: 2
q.c: 10, q.d: 20

ob1.a: 3, ob1.b: 4
ob2.c: 30, ob2.d: 40

ob3.e: 5, ob3.f: 6
```

자, 한 예제에 모두 담아 보았습니다. *p*와 *ob1*을 만드는 부분은 가장 기초적인 구조체 사용법입니다. 여기서의 *a*와 *b*를 **멤버**(**멤버 변수**)라 합니다.

*q*와 *k*를 만드는 것은 조금 특별해 보입니다. 이렇게 구조체를 만들고, 중괄호 뒤에 바로 변수를 선언할 수도 있습니다.

마지막으로 *ob3*을 만드는 것은 많이 이상해 보입니다. 하지만 이는 문법에 맞습니다. 이렇게 이름 없는 자료형으로 구조체를 만들 수도 있습니다.

멤버의 초기화는 어떻게 할까요?

```
struct st1 {
    int a = 0, b = 0;
};
```

아아, 스튜핏! 이런 방법은 통하지 않습니다! 귀찮게스리. 그쵸?

```
#include <stdio.h>
struct st1 {
    int a, b;
};
int main() {
    struct st1 a = { 0, 0 }, b = { 1, 2 };
    printf("a.a: %d, a.b: %d\n", a.a, a.b);
    printf("b.a: %d, b.b: %d\n", b.a, b.b);
    return 0;
}
```

> a.a: 0, a.b: 0
> b.a: 1, b.b: 2

이렇게 ***{ 중괄호 }***를 사용해 각 변수에 대해 하나하나 초기화를 해 주는 방법밖에 없습니다.

구조체 변수의 크기를 보겠습니다.

```
#include <stdio.h>
int main() {
    printf("%d %d %d %d\n",
    sizeof(struct{ int a; }),
    sizeof(struct{ int a, b; }),
    sizeof(struct{ int a, b, c; }),
    sizeof(struct{ int a, b, c, d; }));
    return 0;
}
```

> 4 8 12 16

그냥 평범하게 각 멤버들 크기의 합으로 보이는 것 같습니다만, 실은 아닙니다. 속았죠?

```
#include <stdio.h>
int main() {
    printf("%d %d %d %d\n",
    sizeof(struct{ char a; int k; }),
    sizeof(struct{ char a, b; int k; }),
    sizeof(struct{ char a, b, c; int k; }),
    sizeof(struct{ char a, b, c, d; int k; }));
    return 0;
}
```

> 8 8 8 8

*5 6 7 8*이 나와야 할 것 같은데, *8 8 8 8*이 나옵니다! 좀 이상하지 않나요?

컴파일러는 최대한 좋은 프로그램을 만들어 주려 노력합니다. 과거에는 메모리 자원이 모자랐기 때문에 메모리를 아끼는 것이 가장 좋은 것이라 생각되었지만, 오늘날에는 메모리 자원이 아주 풍족하기 때문에 조금이라도 더 빠른 프로그램이 좋은 평가를 받습니다.

자료형을 설명하며, 명령어의 길이에 맞는 자료형이 가장 빠르다고 한 것을 기억하나요? 이 때문에, 컴파일러는 메모리를 조금 희생하는 한이 있더라도, **구조체의 크기를 멤버 변수 중 가장 큰 크기를 갖는 것에 맞춥니다**. 이렇게요.

```
#include <stdio.h>
int main() {
    printf("%d\n", sizeof(struct{ char a; long long k; }));
    return 0;
}
```

> 16

이런 것을 **바이트 패딩**이라 부릅니다. *#pragma pack*을 사용해 이런 최적화를 할 것인지 말 것인지 결정할 수 있지만, 여기에서는 다루지 않겠습니다.

전처리기

전처리는 컴파일 전에 코드의 내용을 바꿔주는 것을 말합니다. 이러한 동작은 **전처리기**가 수행합니다. ***#*으로 시작하는 문장**들은 전처리기를 위한 것으로, *#include*, *#pragma*, *#define* 등이 있습니다.

***#include* ◇**는 단어 그대로, **◇ 안의 것을 포함**시키라는 뜻입니다. 예를 들어, 우리가 주구장창 써오던 *#include* 〈*stdio.h*〉는 정해진 위치의 *stdio.h* 헤더 파일을 포함시키라는 뜻입니다. 이렇게 함으로서 우리는 ***stdio.h*의 함수와 객체 등을 사용할 수 있게 됩니다**. *stdio.h* 이외에도 *stdint.h*(고정된 크기의 자료형), *setjmp.h*(함수를 넘어 점프, 정확히 말하면 함수 호출 스택의 내용을 저장해 두었다가 복구), *memory.h*(*memcpy*와 같이 메모리에 관련), *math.h*(삼각 함수나 로그 함수같이 수학에 관련), *string.h*(문자열 관련) 등 여러 헤더 파일들이 있습니다. 프로그래머가 **직접 헤더 파일을 작성**하는 것도 가능합니다.

#pragma*는 컴파일에 관련된 중요한 무언가를 수정**하기 위해 사용합니다. 아까 잠시 말한 ***#pragma pack이나 헤더 파일을 한 번만 포함하게 해 주는 ***#pragma once***같은 것이 있습니다.

#define*은 치환 매크로**입니다. 말 그대로 내용을 바꿔줍니다. ***#define 식별자 바꿀 내용의 형식으로 사용합니다.

```
#include 〈stdio.h〉
#define MAX 128
int main() {
```

```
    int a[ MAX ][ MAX ] = { 0, };
    printf("%d\n", MAX);
    return 0;
}
```

이런 식으로, 프로그램 상에서 자주 쓰이는 수에 사용합니다. 만약 한 번에 수정할 일이 생겼을 때, *MAX*를 정의한 줄만 바꿔주면 되기 때문에 무척 유용합니다. 매크로 이름은 관습적으로 **대문자**를 사용하며, 매크로는 *const*와는 다르게 **컴파일 타임에 상수로 취급**됩니다.

```
#include <stdio.h>
#define add(a, b) a + b
int main() {
    printf("%d\n", add(3, 5));
    return 0;
}
```

> 8

별 문제가 없어 보이는 것 같기도 합니다. 이 코드는 다음과 같이 치환될 것입니다.

```
#include <stdio.h>
int main() {
    printf("%d\n", 3 + 5);
    return 0;
}
```

그리고, 제 생각에는 조만간 대참사를 일으킬 것 같군요.

```
#include <stdio.h>
#define add(a, b) a + b
int main() {
	printf("%d\n", add(3, 5) * 8);	//절대 64가 아닙니다!
	return 0;
}
```

> 43

이해할 수 없다면, 위의 코드가 어떻게 치환되는지 한 번 봅시다.

```
#include <stdio.h>
int main() {
	printf("%d\n", 3 + 5 * 8);	//절대 64가 아닙니다!
	return 0;
}
```

매크로 사용 시에는 꼭! **(괄호)**를 사용해 줍시다.

```
#include <stdio.h>
#define add(a, b) (a + b)
int main() {
	printf("%d\n", add(3, 5) * 8);	//이제 64가 되었습니다!
	return 0;
}
```

> 64

하지만 이런 문제는 피해갈 수 없습니다.

```
#include <stdio.h>
#define square(p) (p * p)
int main() {
    int a = 10;
    printf("a의 제곱은 %d\n", square(a++));
    printf("%d\n", a);
    return 0;
}
```

> a의 제곱은 110
> 12

이렇게 치환되기 때문이지요.

```
#include <stdio.h>
int main() {
    int a = 10;
    printf("a의 제곱은 %d\n", a++ * a++);
    printf("%d\n", a);
    return 0;
}
```

스스로 연산자 우선순위를 찾아보며 이유를 찾아 보도록 합시다.

이런 골치아픔으로부터 벗어나는 가장 좋은 방법은 매크로 대신 **인라인 함수**를 사용하는 것입니다. 인라인 함수를 사용하면 경우에 따라서는 **성능상의 이득**을 얻을 수 있는 경우도 있지요. 하지만 매크로는 인라인 함수보다 좀 더 유연합니다.

```
#include 〈stdio.h〉
#define FOR(i, a) for(int i = 0; i 〈 a; ++i)
int main() {
    FOR(i, 3) {
        FOR(j, 4)
            putchar('*');
        putchar('\n');
    }
    return 0;
}
```

매크로의 남용은 다른 사람뿐 아니라 자신도 읽기 어려운 코드를 만듭니다. 적당히 사용하도록 합시다.

함수가 그렇듯 매크로에도 **가변 인자**라는 것이 있습니다. 하지만 어렵기 때문에, 여기까지만 하도록 하겠습니다.

묵시적 형변환

자, 다음 코드에서 뭔가 이상한 점을 찾아보세요.

```
#include <stdio.h>
int main() {
    int a = 6.5;
    printf("%d\n", a);
    return 0;
}
```

> 6

자, 어떻게 *int* 자료형에 *6.5*를 담았을까요?

C 언어에서는 서로 맞지 않는 자료형 간의 연산 중 일부에 대한 대책을 세워놓았습니다. 이것이 바로 **묵시적 형변환**입니다.

묵시적 형변환에는 몇 가지 규칙이 있습니다.

첫째, **정수 자료형에 실수 자료형을 담으면 소수점 아래는 버려집니다**. 위에서 본 것이 그 예시 중 하나이지요. 단,

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

```
    printf("%d\n", 6.5);
    return 0;
}
```

이건 조금 곤란합니다. 이건 *printf*의 작동 방식과 관련된 부분이기 때문에, 지금은 이렇게 하면 안된다는 것만 알아두세요.

둘째, **실수 자료형에 정수 자료형을 담으면 *.0*이 생깁니다**.

```
#include <stdio.h>
int main() {
    float a = 6;
    printf("%f\n", a);
    return 0;
}
```

> 6.000000

셋째, **작은 자료형에 큰 자료형의 값을 대입하면 하위 *n*바이트만 가져옵니다**.

```
#include <stdio.h>
int main() {
    short a = 0x12345678;
    printf("%hX\n", a);
    return 0;
}
```

> 5678

넷째, **양쪽 다 *int*보다 작은 정수형일 경우 연산의 결과는 *int*형입니다.**

```
#include <stdio.h>
int main() {
    short a = 10, b = 5;
    printf("%d...%d\n", a + b, sizeof(a + b));
    return 0;
}
```

> 15...4

다섯째, **양쪽 다 정수형이거나 양쪽 다 실수형일 경우 연산의 결과는 둘 중 더 범위가 넓은 자료형이 됩니다.** 참고로 **상수 뒤에 *ll*을 붙이거나 *u*를 붙이면 각각 *long long*과 *unsigned***입니다. 조합해서 *ull*을 만들면 *unsigned long long*이 되고, **아무것도 없으면 그냥 *int***입니다.

```
#include <stdio.h>
int main() {
    printf("%d...%d\n", 10ll + 10, sizeof(10ll + 10));
    return 0;
}
```

> 20...8

여섯째, **부호 없는 정수형과 부호 있는 정수형일 경우 연산의 결과값은 부호 없는 정수형**이 됩니다.

```
#include <stdio.h>
int main() {
	unsigned int a = 0;
	for(int i = 0; i < a - 1; ++i)
		printf("*\n");
}
```

> *
>
> *
>
> *
>
> *
>
> ...

산술 오버플로우로 인해 *a - 1*의 값은 *4294967295*가 되고, 결국 *4294967295*개의 *****이 출력됩니다. *a*가 *int*였다면 한 개도 출력되지 않았을 텐데 말이죠.

말 나온 김에, 컴퓨터에서의 음수와 오버플로우에 대해서도 설명하죠. 컴퓨터는 음수를 저장할 때, '**2의 보수**'라는 방법으로 저장합니다. **모든 비트에 NOT연산을 취한 뒤, *1*을 더해주는 거죠**. 이렇게 하면 양수와의 **덧셈도 자연스럽게** 가능하고, **뺄셈도 쉬우니까요**.(*a - b*는 *a + (-b)*로 바꿔서 계산). **맨 오른쪽(상위) 비트는 부호 식별용**으로 사용됩니다.

주유소에서 기름을 100만원어치 넣으면 계기판이 999997원, 999998원, 999999원 하고 올라가다 000000원이 되는 것을 상상해 본 적이 있습니까? 이런 것을 '**오버플로우**'라고 합니다. *char* 자료형을 생각해 봅시다. 현재의 값은 $0111\ 1111_{(2)}$입니다. 여기서 *1*을 더하면 $1000\ 0000_{(2)}$입니다. 아, 양수가 너무 커지다 보니 음수가 되었습니다. *unsigned* 였다면 $1111\ 1111_{(2)}$에서 커지다 못해 $0000\ 0000_{(2)}$이 되었겠지요.

미처 하지 못한 말들

다루지 않은 것

사실 이 책만 해도 C 언어의 꽤 많은 부분을 담고 있습니다만, 아직 우리에게는 배우지 않은 C언어의 다양한 기능들이 있습니다. 앞으로 C언어를 더 공부하고 싶은 사람들을 위해, '공부하면 좋은 것들'을 담아 보았습니다.

가변 인자

혹시 지금껏 *printf* 함수와 *scanf* 함수를 써오며 이상함을 느끼지 않았나요? 어떻게 이 두 함수는 받는 인자의 개수가 달라질까요? 어라? *%d*의 개수와 인자의 개수가 맞지 않아도 오류가 나지 않잖아? 잠깐, 나는 분명 *printf("%d", "BOY");*라고 했는데 왜 무언가 출력되지?

그 답은 가변 인자에 있습니다. 가변 인자를 공부해 두면 어셈블리 언어에서 프로시저(함수)에 인자를 전달하는 과정을 쉽게 이해할 수 있습니다.

longjmp

열심히 공중 재귀를 돌다 재귀를 탈출하고 싶을 때 어떤 방법을 사용하나요? 아마 다들 *if*와 *return*을 열심히 사용할 것입니다. 하지만 이 길고 긴 함수 호출 스택을 훅 바꿔버리는 마법이 있습니다!

공용체

사실 많은 C 언어 프로그래밍 책에서 공용체를 다루고 있지만, 컴퓨터의 성능이 좋아지고 메모리의 용량이 크게 증가한 요즘, 공용체는 잘 사용되지 않습니다. 그렇기 때문에 C발라먹기에서는 이를 의도적으로 배제하였지만, 혹시 'C 언어의 모든 것을 알아야겠다!'고 한다면 공용체를 공부하십시오.

수많은 라이브러리

C 언어에서는 자주 사용하는 기능들을 모아 헤더 파일 형태의 라이브러리로 제공하고 있습니다. 다른 언어들만큼은 아니지만, C 언어의 기본 라이브러리는 무척 방대합니다. 또 기본 라이브러리 이외에도 여러 라이브러리가 있는데, 여기에서 모두 다룰 수는 없으니 따로 공부해 보시기 바랍니다.

인라인 어셈블리

'C 언어만으로는 속도가 모자라!'라고 생각한다면, C 언어 코드 중간에 어셈블리 코드를 박을 수도 있습니다. 물론 완전한 어셈블리 언어는 아닙니다. 하지만 요즘은 컴파일러가 무척 좋기 때문에 권장하지 않고, 모든 환경에서 정상 작동 하리라는 보장도 없습니다. 인라인 어셈블리를 쓰고 싶다면 한 마디만 기억하십시오.

> 코딩보다 중요한 것은 흐름이다.

이 말은 아래의 비트 연산이나 포인터 꼼수에도 적용됩니다.

비트 연산

컴퓨터는 수를 2진수로 저장합니다. 이 수를 비트 단위로 조종함으로서 특정 작업들을 쉽게 할 수 있는 경우가 있는데, 이 조종 작업을 비트 연산이라 합니다. 비트 연산을 적극적으로 도입하면 메모리를 아끼고 속도를 단축시킬 수 있지만, 자칫하면 찾기 힘든 버그가 발생하거나 남들이 읽기 힘든 코드가 되므로 주의해 주세요.

비트필드 구조체

비트필드 구조체를 사용하면 멤버들의 크기를 비트 단위로 지정할 수 있습니다. 공용체와 같이 잘 쓰이지 않지만, 일부 저수준 작업에서 필요한 경우도 있습니다.

파일 입출력

지금까지는 *printf*와 *scanf*를 사용한 콘솔 입출력만을 했지만, C 언어는 파일 입출력을 지원합니다. 텍스트 파일에서 글자를 읽어오는 것뿐만 아니라 다양한 종류의 파일들을 읽고 쓸 수 있습니다. 다만, 환경에 따라 달라질 수 있는 요소들도 많고, 복잡한 부분도 많아 여기에서는 생략했습니다.

멀티 스레드

멀티 스레딩을 통해 작업의 병렬화가 가능합니다. 다만, 메모리를 공유할 경우 주의를 기울여야 하며 작업간의 종속성이 없을 경우에만 멀티 스레딩을 통한 병렬화가 가능합니다. 또 멀티 스레딩을 하기 위한 오버헤드도 감안할 필요가 있습니다. 많은 온라인 알고리즘 트레이딩 사이트에서는 멀티 스레딩을 금지하고 있습니다.

동적 할당

배열은 우리가 처음 선언할 때 지정한 크기 이상으로 사용하지 못합니다. 그런데, 경우에 따라서는 더 큰 크기의 배열이 필요해질 수도 있습니다. 이럴 때 사용하는 것이 동적 할당입니다. 하지만 잘못 사용할 경우 허상 포인터나 메모리 누수 등의 문제가 일어날 수 있고, 캐시 효율에도 악영향을 끼칠 수 있으니 주의해서 사용해 주세요.

더 공부할 것

이 책을 모두 읽었다면 컴퓨터의 다른 부분들에 대해서도 공부하고 싶어졌을 것입니다. 다른 언어에 대해 궁금해졌을 수도 있고, C 언어를 깊게 파 보고 싶기도 할 것입니다. 여러분을 위해 몇 가지 공부해봄직한 것들을 간단하게 소개하겠습니다. 추천 도서에서 원서는 의도적으로 배제하였습니다.

알고리즘 & 자료구조

알고리즘과 자료구조는 지금까지 배운 프로그래밍 언어에 수학을 도입하여 언어를 좀 더 효과적으로 사용할 수 있게 해줍니다. 좋은 알고리즘을 사용하면 같은 동작을 해도 더 효율적으로(적은 메모리 사용과 실행 시간) 작동합니다. 많은 언어들에서 기초적인 알고리즘과 자료구조를 제공하지만(대표적으로 C++의 STL), 구현이 되어 있지 않은 알고리즘도 많고, 또 이미 있는 것이라 해도 원리를 알고 사용하는 것과 그렇지 않은 채로 사용하는 것에는 큰 차이가 있기 때문에 한 번 쯤은 공부해 두는 것이 좋습니다.

알고리즘을 공부하고 싶다면 『알고리즘 문제 해결 전략』(구종만, 인사이트)를 추천합니다. 이 이외에 백준 온라인 저지(https://www.acmicpc.net) 등에서 알고리즘을 배우고 문제를 풀 수 있습니다. 이러한 온라인 알고리즘 트레이닝 사이트 사용에 팁을 조금 주자면, 실행 시간을 줄이겠다고 과도하게 스레드를 사용할 경우 일시적인 밴을 당할 수 있기 때문에 단일 스레드만을 사용하길 바랍니다.

컴퓨터 구조

이 언어 밑에서 어떤 작업들이 이루어지고 있는지, CPU는 어떻게 동작하는지 등을 알고 싶다면 컴퓨터 구조에 대해 공부하는 것이 좋습니다. 레지스터, 메모리, 캐시, 명령어 등에 대한 개념을 익히고 이를 코딩에 적용한다면 훨씬 효과적으로 움직이는 프로그램을 작성할 수 있을 것입니다. 컴퓨터 구조를 공부하고 싶다면 『컴퓨터 구조 및 설계』(David A. Patterson & John L. Hennessy, 한티미디어)나 『컴퓨터 시스템』(Randal E. Bryan & David R. O'Hallaron, 피어슨에듀케이션코리아)을 추천합니다. 다만 한 가지 말해두자면 학부 과정의 내용들을 담고 있으니 어렵다고 좌절하지는 마시기 바랍니다.

디자인 패턴

디자인 패턴은 오랫동안 연구되고 사용된 코드들의 패턴으로, 프로그램의 구조를 깔끔히 하는 데 도움이 됩니다. Builder, Visitor 등의 패턴이 있으며, 지금도 많은 디자인 패턴들이 만들어지고 있습니다. 디자인 패턴을 공부하면 오랜 시간 코딩해야만 얻을 수 있는 노하우들을 단기간에 얻을 수 있습니다.

C

만약 이 책을 읽은 후 자신이 C 언어를 마스터했다고 생각한다면 천만의 말씀, 만만의 콩떡입니다. '다루지 않은 것'에도 없는 C 언어의 수많은 기능과 그로부터 나온 수많은 꼼수들은 감히 그 전체 크기조차 헤아

리기 쉽지 않습니다. 좀 더 심화적으로 C 언어를 공부하고 싶다면 『C 기초 플러스』(스티븐 프라타, 성안당)와 『이것이 C 언어다』(서현우, 한빛미디어)를 추천합니다. 이 이외에 Cpp Reference 등에서도 C 언어를 공부할 수 있습니다.

C++

C++은 1983년, 비아네 스트로스트룹에 의해 만들어진 언어로, C언어에 객체 지향 프로그래밍의 개념을 얹은 것입니다.

C++은 여러 차례의 개정을 거쳐 무척 크고 복잡한 언어가 되었습니다. C언어의 속도와 고급 언어들의 생산성 중 아무 것도 놓지 않았기 때문에 C++은 생산성 대비 속도를 생각해 보았을 때 아주 좋은 언어입니다. 특히 C++의 꽃이라 불리는 템플릿과 STL은 다른 언어에서는 찾아보기 힘든, C++의 자랑 중 하나입니다. 하지만 복잡한 문법들과 많은 기능들, 그리고 끊임없이 생겨나는 꼼수들 덕분에 C++에는 '평생 언어'라는, 웃을 수 없는 별명이 붙어 있습니다. 컴퓨터를 한계까지 쥐어짜 내고 싶은 사람이라면 배워봄직한 언어입니다.

C++은 게임이나 이미지 프로세싱, 압축 등 성능과 생산성 중 어느 것도 놓칠 수 없는 분야들에서 사용됩니다. 운영체제 쪽은 아직 C가 대부분이지만, 가끔씩 C++을 사용하기도 합니다. 꼭 C++로 전부 만들지 않더라도, 성능이 필요한 부분은 C++로 만들고 나머지 부분은 다른 고생산성 언어로 만들어 붙힐 수도 있습니다.

C++을 공부하고 싶다면 『C++ 기초 플러스』(스티븐 프라타, 성안당)와 『이것이 C++이다』(최호성, 한빛미디어)를 추천합니다. 이 이

외에 Cpp Reference나 C plus plus, MSDN C++ 언어 참조등에서도 C++을 공부할 수 있습니다.

Ruby

Ruby는 1995년, 마츠모토 유키히로(Ruby 사용자 사이에서는 '마츠'라는 별명으로 불립니다)에 의해 만들어진 언어로, C/C++과는 다르게 컴파일러 대신 인터프리터를 사용합니다. 인터프리터를 사용하면 컴파일러를 사용할 때 보다 쉽게 프로그램을 수정할 수 있지만, 속도 면에서는 컴파일러를 사용하는 것보다 조금 느릴 수 있습니다. 루비는 2.6.0 버전부터 JIT 컴파일을 지원하기 때문에 다른 스크립트 언어들에 비하면 속도가 빠른 편입니다.

C++가 C언어에 객체 지향 프로그래밍의 개념을 얹은 것이라면, Ruby는 처음부터 객체 지향 프로그래밍 언어로 설계되었습니다. 따라서 문법도 간단하며, 언어 자체를 익히는 것은 C 언어보다 훨씬 쉽습니다. 이미 아는 언어가 있다면 무척 빠르게 익힐 수 있겠지요. 하지만 Ruby는 하드웨어로부터 멀리 떨어진, 다시 말해 고도로 추상화 된 언어이기 때문에 Ruby만으로 컴퓨터를 깊게 익히기는 힘듭니다.

C/C++가 운영체제와 게임 등 성능이 필요한 분야에서 주로 사용되는데 비해, Ruby는 생산성이 중요시되는 곳에서 주로 사용됩니다. '속도는 모르겠고 일단 띄우고 본다'라는 의미지요. 특히 Ruby on Rails와 함께 웹 개발에서 인기가 많습니다. 비슷한 목적을 가진 언어로는

Python, Lua 등이 있으나, 객체 지향 프로그래밍을 잘 익히고 싶다면 Ruby를 배울 것을 추천합니다. 물론 위의 두 언어도 객체 지향 언어입니다.

Ruby를 공부하고 싶다면 『프로그래밍 루비』(데이브 토머스 & 채드 파울러 & 앤디 헌트, 인사이트)를 추천합니다. 이 이외에 루비 한국어 문서에서도 루비의 간단한 기능들을 학습할 수 있습니다.

C#

이름에서 알 수 있듯, C#은 C 언어로부터 파생된 언어입니다. 2000년, 마이크로소프트에서 개발되었으며, C++++에서 ++++를 적절한 모양으로 배치해 #이 되었다고 합니다.

C++이 C에 객체 지향을 그대로 얹어놓은 것이라면 C#은 Java처럼 적극적으로 객체 지향을 도입하였습니다. C++보다 쉽다는 평가를 얻고 있지만, 최근 들어 기능이 많이 추가됨에 따라 난이도가 상승중인 언어입니다.

C#은 독특한 컴파일 방식을 가지고 있습니다. 이를 알기 위해서는 컴파일러의 구조를 약간 알아야 합니다. 컴파일러는 대개 프론트엔드 - 미들엔드(옵티마이저) - 백엔드의 구성을 갖고 있습니다. 이러한 구조를 갖게 되면 언어의 수와 아키텍처 수의 곱만큼 컴파일러가 필요하게 됩니다. 이를 해결하는 방법은 고급 언어와 기계어 사이의 중간 언어를 만드는 것입니다. 컴파일러는 고급 언어를 중간 언어로 번역하고, 가상 머신은 중간 언어를 기계어로 번역합니다. 이러한 구조를 취하게 되면 언어가 1개 추가되어도 1개의 프론트엔드 컴파일러만이 더 필요하고,

아키텍쳐가 1개 추가되어도 1개의 백엔드 컴파일러만이 필요합니다. 물론 네이티브 기계어로 번역하는 것보다는 실행 속도가 느리지만, 컴파일러를 만드는 데 필요한 노력이나 이식성 등을 생각해 보면 이 정도의 속도 손실은 감안해야 합니다. 최근에는 JIT 컴파일을 도입하여 속도를 향상시키는 추세입니다. 이러한 구조는 JVM(Java, Kotlin, Scala, Clojure 등), .NET Framework(C#, Visual Basic .NET 등), LLVM(Clang C++, Swift 등)에서 볼 수 있습니다.

C#이 사용하는 가상머신은 .NET Framework입니다. C#의 컴파일러는 CIR 코드를 내놓고, 이는 .NET CLR에 의해 기계어로 변환됩니다. .NET 프레임워크를 사용하는 다른 언어에는 VB .NET, C++ .NET 등이 있습니다.

C#은 마이크로소프트에서 만든 언어답게 Windows용 프로그래밍 제작에 특화되어 있습니다. 하지만 요즘, C#을 Windows가 아닌 다른 플랫폼에서도 사용할 수 있게 하려는 시도들이 있으니 기대해 볼 만 합니다. 또, C#만으로는 속도가 모자랄 때, 속도가 필요한 부분은 C++로 만들고, 나머지 부분들은 C#으로 만드는 방법은 요즘 큰 인기를 얻고 있는 프로그래밍 방법 중 하나입니다.

Swift

Swift는 2014년, 애플에서 개발한 프로그래밍 언어로, 기존의 Objective-C를 대체해 OS X, iOS용 프로그래밍 언어로 사용됩니다. 애플 기기용 앱을 만들고 싶다면 반드시 익혀야 하는 언어입니다.

Swift는 LLVM 위에서 실행되는데, 가상 머신 위에서 돌아가는 다른 언어들과는 다르게 IR(LLVM 중간 언어)와 Swift 코드 사이에 중간 언어가 하나 더 있습니다. 이를 통해 Swift는 많은 최적화 기회를 얻습니다.

현재는 애플 기기용으로밖에 사용하지 못하지만 곧 Windows와 Android에서도 사용 가능해질 예정입니다.

Kotlin

Kotlin은 2011년, JetBrains에서 개발한 언어로, JVM에서 돌아갑니다. LLVM을 사용한 컴파일 또한 지원합니다.

Kotlin은 Java보다 간단한 문법들과 고급 기능들을 가지고 있어 큰 인기를 얻고 있습니다. 또 기존의 Java 프로그래머들이 배우기 쉽다는 점 덕분에 벌써 많은 회사가 Kotlin을 도입했습니다. 2017년에는 구글 Android의 공식 언어로 지정되기도 하여 앞으로도 상승세일 예정입니다.

Lisp

Lisp는 1958년, 존 매카시가 개발한 프로그래밍 언어로, FORTRAN 다음으로 오래된 언어입니다. 사실 Lisp는 어느 한 언어만을 이야기한다기 보다는 표준인 Common Lisp와 그 사투리들(Scheme, Clojure, Emacs Lisp 등)을 모두 가리키는 표현입니다(Lisp-family). 다른 언어들이 여러 가지 괄호를 사용하는 데 비해 Lisp에서는 오직 ()만을 사용

합니다. 또, Lisp에서는 사칙 연산 같은 모든 것이 함수로 취급되며, 때문에 리스프 코드를 보면 괄호가 무척 많습니다. 예를 들어 3 * 15 + 20 / 10과 같은 간단한 식도 리스프에서는 (+ (* 3 15) (/ 20 10))라고 씁니다.

이름에서 알 수 있듯 리스트를 다루기 쉬우며 함수를 일급 객체로 취급한다는 점, 그리고 코드와 데이터를 자유롭게 바꿀 수 있고 *eval* 함수가 존재한다는 점 등 덕분에 메타 프로그래밍이 쉬우며, 인공지능 같은 분야에 주로 사용됩니다. 그 이외에 AutoCAD나 Emacs 등에도 사용됩니다.

당시로서는 혁신적이었던 프로그래밍 개념인 Garbage Collection이나 JIT 등을 도입했으며, 이는 현대의 다른 프로그래밍 언어들에도 영향을 주었습니다.

Haskell

Haskell은 1997년 개발된 순수 함수형 프로그래밍 언어로, 앞에서 소개한 다른 언어들과는 다른, 커링, 느긋한 계산 등의 특징들을 갖고 있습니다.

기존의 프로그래밍 언어들은 변수와 조건 분기를 사용합니다. 즉, 튜링 머신에 그 근간을 두고 있습니다. 하지만 Haskell과 같은 순수 함수형 프로그래밍 언어들은 이것들을 사용하지 않고, 람다 대수를 사용합니다. 이 둘은 동치임이 수학적으로 증명되어 있습니다.

이러한 프로그래밍 언어에서는 변수를 사용하지 않기 때문에 모든 함수는 상태를 갖지 않는, 순수한 수학의 함수이고(이를 참조 투명성이라 합니다), 명령어들의 순서는 아무런 의미를 갖지 않습니다. 원칙적으로는 말입니다. 실제로는 I/O를 위해 약간의 Side Effect가 필요합니다.

이러한 참조 투명성으로 인해 순수 함수형 언어들은 병렬 작업(멀티 스레딩)이 쉽습니다. 단일 코어의 성능을 올리기 힘든 요즘, 순수 함수형 언어들이 다시 주목받고 있습니다.

Haskell의 다른 특징에는 타입 클래스와 모나드가 있습니다. 타입 클래스는 타입 변수를 사용해 한 타입을 인자로 받아 다른 타입을 내보내는 것으로, C++의 template과 비슷합니다. 또, 이 타입 변수에 들어갈 수 있는 타입들을 제한할 수도 있습니다. 모나드는 범주론의 개념으로, '내부 함자 범주의 모노이드 대상'으로 정의되어 있습니다. 물론 이렇게 써 놓으면 알아볼 수도 없을 뿐만 아니라 프로그래밍에도 도입할 수 없기 때문에, Haskell에서 실제로 사용하는 모나드는 조금 다릅니다.

아까 말했듯, Haskell의 모든 함수는 상태(statement)를 갖지 않는, 순수한(pure) 함수입니다. 순서에 의미가 없다고 했으니, I/O 작업은 불가능하겠지요. 그런데, 여기서 함수의 인자 중에 statement가 있고 함수의 반환값이 statement에 대한 정보를 포함하고 있다면 어떻게 될까요? 좀 더 정확히 말하자면, 함수가 상태와 결합하는 것입니다. 지금까지 말한 것을 종합하자면, Haskell에서는 함수가 statement를 받고 statement를 뱉는 과정을 반복하며 함수의 순차적 시행을 흉내 냅니다. 하지만 여전히 함수는 순수하지요.

사실 모나드는 I/O를 위해 만들어진 것으로, 초기 Haskell에는 존재하지 않았습니다. 따라서 Haskell을 공부하기 위해 반드시 모나드를 익힐 필요도 없고, *do*같은 문법으로 좀 더 쉽게 순차적 시행을 할 수도 있습니다. 너무 겁먹지는 말고, 한 번 공부해 보는 것도 나쁘지 않습니다.

Rust

Rust는 2015년, 모질라 재단에서 개발한 언어로, 현재의 C++를 대체하려 하고 있습니다. Rust는 Garbage Collection 없이 메모리 안전성을 제공합니다. 이는 값의 대입이 소유권의 이전을 의미하게 함으로써 달성합니다. 즉, 값은 이름에 종속적이며, 컴파일 타임에 소유권의 모든 이전을 추적할 수 있습니다. 또, Rust에서 변수는 기본적으로 불변, 즉 수정이 불가능한데, 이렇게 불변을 권장하고 공유를 금지함으로써 Rust는 멀티 스레딩에서 큰 이점을 갖고 있습니다.

Rust는 아직 신생 언어이지만, 그 독특한 설계 덕분에 2017년 Stack Overflow 설문조사에서 선호 언어 1위를 하는 등 큰 인기를 끌고 있습니다.

HTML

HTML은 마크업 언어로, 웹 페이지를 만들 때 사용합니다. 웹 페이지를 만드는 데 관심이 있다면 반드시 알아야 하는 것 중 하나입니다.

Linux

Linux는 컴퓨터 커널의 일종으로, 이를 사용해 만든 운영체제들을 GNU/Linux라 합니다. 이들 운영체제는 원한다면 자유롭게 수정해 사용할 수 있습니다.

Windows와 구분되는 GNU/Linux만의 가장 큰 특징은 바로 CLI입니다. 물론 Windows에서도 cmd를 사용하여 비슷한 작업이 가능하지만, 수준이나 편의성을 생각해 보았을 때 GNU/Linux 쪽의 bash쪽이 훨씬 낫습니다. 또한, 많은 개발 환경들이 Linux 전용으로 만들어져 있기 때문에, 본격적인 개발을 하고 싶다면 Linux를 배우는 것도 좋다고 생각합니다.

WinAPI

WinAPI는 C언어를 위한 Windows용 API로, 하드웨어나 메시지, 커널 등에 관련된 사항들을 쉽게 처리할 수 있게 도와줍니다. Windows용 프로그램을 만들기 위해서는 반드시 알아야 하는 것의 하나입니다.

마치며

사실 언어는 도구에 불과합니다. 물론 더 빠르거나 쓰기 편하고, 또 사고의 방향 자체를 바꿔주는 언어들도 있지만, 가장 중요한 것은 프로그래머입니다. 여기에 나온 내용을 모두 외웠다고 해서 자만하지 말고 키보드를 두드려 프로그램을 만드십시오. 만들다 보면 이론 이상으로, 손과 마음으로 알게 되는 것들이 있습니다. 프로그래밍 연습은 자신만의 테크닉을 만들어가는 과정입니다.

한 줄을 아름답게 작성하는 것도 중요하지만, 전체 구조를 잘 잡는 것은 더 중요합니다. 특히 큰 프로그램에서 그렇습니다. 이를 위해 OOP(객체 지향 프로그래밍)이나 순수 함수형 프로그래밍 등이 등장하였습니다. C언어에서도 이러한 패러다임을 흉내낼 수는 있지만, C++이나 Ruby처럼 처음부터 그러한 패러다임을 지원하는 언어에는 미치지 못합니다. 물론 이러한 언어들은 고급 기능을 지원하기 위해 C보다는 속도가 느립니다.

어느 정도 경지에 이르기 전까지는 책이나 글을 보는 것보다는 코드를 짜는 것이 프로그래밍 실력 향상에 도움이 됩니다. 그 이후에 컴퓨터의 구조를 공부한다면 하드웨어에 최적화되고 견고한 구조를 가진 코드가 손에서 자연스럽게 나올 것입니다. 이 책을 모두 읽은 여러분이 할 일은 당장 IDE를 켜는 일입니다.

마지막으로 이 책을 끝까지 읽어낸 여러분께 박수를 드립니다. 또 조촐한 책이지만 이 책이 나올 수 있게 도와주신 분들께 감사의 인사를 드립니다.

Goodbye, world!

폰트 라이센스

- 나눔 바른고딕
  Copyright (c) 2010, NAVER Corporation (http://www.nhncorp.com)
- 나눔 명조
  Copyright (c) 2010, NAVER Corporation (http://www.nhncorp.com)
- KoPub돋움체 Medium
  문화체육관광부(http://www.mcst.go.kr), 한국출판인회의(http://www.kopus.org)에 저작권이 있습니다.
- KoPub바탕체 Light
  문화체육관광부(http://www.mcst.go.kr), 한국출판인회의(http://www.kopus.org)에 저작권이 있습니다.