목차

5. 내용그래밍

1. 메모리 배치
2. 대형 프로그램의 구성
3. 정적 외부 변수/함수
4. 추상 자료형

6. 입출력 함수

1. 스트림
2. 입출력 함수들

7. 파일

1. 파일 입출력
2. 파일 위치 지시자

기타

1. 변수 선언 옵션들
2. 가변 인자
3. 진단 코드
4. 신호
5. 프로그래밍 도구
6. 기타

1. 포인터

1. 포인터

1. 메모리 사용

1) 메모리 관리의 단위

운영체제는 메모리를 1바이트 단위로 주소를 부여하여 관리함.

2) 메모리 사용

메모리를 사용하려면 세 가지가 필요함.

1. 메모리 할당. 2. 시작 주소. 3. 사용할 크기. (또는 끝 주소.)

3) 일반변수에서의 메모리 사용

1. 정적 메모리 할당.
2. 식별자 사용. -> 이 식별자는 컴파일 시에 시작 주소로 변환됨.
3. 선언문에 작성한 크기 만큼 사용.

4) 포인터에서의 메모리 사용

1. 동적 또는 정적 메모리 할당.
2. 포인터 변수에 저장한 메모리 주소 사용.
3. 포인터 형의 크기 만큼 사용.

+ 메모리

cpu가 직접 접근하여 데이터를 다룰 수 있는 유일한 저장 장치.

메모리는 각 바이트 별로 주소가 붙여진 1차원 배열로 생각할 수 있음.

+ 주소 값을 갖는 수식의 메모리 크기

포인터 변수와 같음. (시스템에 따라 4 바이트 또는 8 바이트)

메모 포함[이1]: 메모리 주소를 다루기 위한 문법.

메모 포함[이2]: 즉, 각 주소는 1바이트 크기의 메모리 공간을 가진다.

메모 포함[이3]: 보통 시작 주소는 그냥 주소라고 하는 것 같다.

메모 포함[이4]: 프로그램 실행 시 실행 파일이 메모리에 재구성되는 이유이다

2. 포인터 변수

메모리 주소를 다룰 수 있도록 하는 변수.

void 형을 제외한 포인터 변수에 값을 배정할 때에는 그 값이 동일한 자료형을 가져야 경고 메시지가 안 뜸.
단, 0은 자료형에 상관없이 배정이 가능함.

1) 포인터 변수의 선언

<자료형> *<변수명>;

자료형 뒤에 *을 붙임.

여러 개의 포인터 변수를 한 선언문에서 선언할 때에는 각 포인터 변수 이름 앞에 *을 붙여 줌.

2) 포인터 변수의 형

포인터 변수가 가지는 메모리의 크기는 시스템에 따라 4바이트 또는 8바이트로 고정임.

포인터 변수 선언 시에 작성하는 자료형은 포인터가 가리키는 대상의 자료형(크기)을 의미함.
포인터 변수 역참조 시에는 포인터 변수의 형에 맞춰 참조 방식을 선택함.

3) 포인터 변수의 사용

1. 포인터 변수의 값을 다루는 것과 2. 포인터 변수가 가리키는 것의 값을 다루는 것으로 나뉨.

1번은 포인터 변수로의 대입을 통해서, 2번은 번지 지정 연산자를 통해서 수행함.

3. & 연산자 (ampersand 앰퍼샌드 연산자)

변수의 주소를 구하는 단항 연산자.

&<변수명>

1) 사용 이유

프로그램과 프로그램의 변수는 실행할 때마다 새로운 메모리 주소를 부여받음.
그래서 포인터 변수에 고정된 상수를 대입해 사용하면 오류가 발생함.

매 실행 때마다 달라지는 변수의 주소를 구하기 위해 & 연산자를 사용할 수 있음.

2) 주의점

& 연산자는 메모리를 할당 받은 객체에만 사용할 수 있음.
스토리지 클래스가 register 인 변수, 상수, 메모리 할당이 되지 않은 수식에는 사용할 수 없음.

메모 포함[이5]: 포인터와 관련된 생각을 할 때는 포인터 메모리 공간과 포인터가 가리키는 메모리 공간을 두 사각형으로 생각하여 그림을 떠올리면 효과적이다. (linked list의 노드를 생각할 때 사용했던 방식.)

메모 포함[이6]: ex.
int *a, *b, *c;

메모 포함[이7]: 단순히 '크기'가 아닌 이유는, 포인터로 특정 변수의 값을 포인트해서 사용할 때, int 형을 읽을 것인지 float 형으로 읽을 것인지와 같은 문제가 생기기 때문이다.

메모 포함[이8]: 주소가 존재하지 않는 것들.
ex. 정수/실수 상수, 문자 등.

참고로, 문자열은 주소가 존재한다.

4. * 연산자 (역참조 연산자) (간접 지정 연산자)

포인터 변수가 가리키는 메모리를 사용한다는 의미의 단항 연산자.

주소 값을 갖는 대상에만 사용할 수 있음.

우->좌의 결합 순위를 가짐.

메모 포함[이9]: ex.
int var;
int *ptr = &var;
*ptr = 0x30; -> var변수에 0x30을 대입.

5. 포인터의 자료형과 대상의 자료형

포인터 변수 선언 시에 작성하는 자료형은 포인터가 가리키는 대상의 크기를 의미함.

1) 포인터 변수의 자료형과 포인터 변수가 가리키는 변수의 자료형이 같은 경우

일반적으로는 두 자료형을 같게 맞춤.

포인터가 특정 주소의 값을 읽을 때, 포인터의 형과 값의 자료형이 일치하지 않으면 오류가 발생함.

메모 포함[이10]: ex. int 형의 값을 float 형의 포인터로 읽을 수 없다.

2) 포인터 변수의 자료형과 포인터 변수가 가리키는 변수의 자료형이 다른 경우.

포인터 변수 자료형의 크기를 가리키는 변수의 자료형보다 작게 하는 경우도 있음.

이 경우 포인터 변수에 저장된 시작 주소부터 포인터 변수의 자료형의 크기 만큼의 정보만을 다룸.
이 경우 주소 연산을 사용하면 포인터 변수의 자료형의 크기를 기준으로 연산됨.

메모 포함[이11]: ex.
int a = 0x123456;
char *ptr = (char *)&a;
printf("%x", *ptr);
-> 이 경우 56만 출력된다. (리틀 엔디언에서)

메모 포함[이12]: ex.
int a = 0x123456;
char *ptr = (char *)&a;
printf("%x", *(ptr + 1));
-> 이 경우 34가 출력된다. (리틀 엔디언에서)

이 경우에도 포인터 변수에 주소를 지정할 때 형변환을 해 줘야 함.

이렇게 사용했을 때 주소 처리가 좀 이상해짐. 나중에 확인해보자.

+ 포인트하다.

포인터 변수에 특정 주소가 저장되어 있으면, 포인터 변수가 해당 주소를 '포인트한다'라고 함.

+ 포인터 변수의 초기화

포인터 변수는 선언할 때나 사용하지 않을 때 NULL(0)으로 초기화해두는 것이 좋음.

6. void 형 포인터 변수

메모 포함[이13]: void : 빈 공간, 비어 있는.

<void> *<변수명>;

정해진 자료형이 없음을 표현하는 자료형.

void 형 포인터에는 자료형에 상관없이 수식을 배정할 수 있음.

1) 사용

형이 없는 포인터를 나타내는 경우.

하나의 포인터로 자료형이 다른 여러 개의 주소를 포인트해야 하는 경우.

2) 주의점

void 형 포인터를 사용할 때(* 연산자 사용 시)는 적절한 형 변환이 반드시 필요함.
-> 역참조 시에는 그 방식을 정해줘야 하기 때문.

(ex.

```
void *v;
int i = 100;
float f = 10.0;
v = &i;
printf("i is %d.\n", *((int *)v));
v = &f;
printf("f is %f.\n", *((float *)v));
```

)

7. 포인터의 포인터 (2차원 포인터)

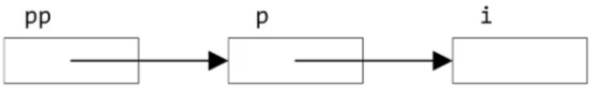
<자료형> **<변수명>;

포인터를 포인트하는 포인터. (포인터 변수의 주소를 저장하는 포인터.)

포인터 변수의 주소 값은 포인터 변수(1차원 포인터)에 저장할 수 없음.
포인터 변수의 주소 값은 포인터의 포인터(2차원 포인터) 변수를 사용해서 다루어야 함.

포인터의 포인터는 선언 시에 *을 2개 작성함.

포인터의 포인터가 가리키는 포인터가 가리키는 곳으로 바로 역참조하려면 **<변수명>으로 작성함.
(역참조 연산자 2번)



8. 포인터 연산 (포인터 주소 연산)

특히 대량의 데이터를 사용해야 할 때 사용함.

1) 포인터와 정수 연산

포인터 변수에 1을 더하거나 빼면 해당 포인터 변수 자료형의 크기만큼 주소가 증가하거나 감소함.
즉, 포인터 변수에 1을 더하거나 빼면 바로 다음 또는 이전 메모리를 사용할 수 있음.

메모 포함[이14]: ex.
int *ptr = (int *) 100;
ptr += 1;
이 경우 ptr에 저장된 시작 주소는 104가 된다.

2) 포인터와 포인터 연산

메모 포함[이15]: 두 포인터의 자료형이 다르다면 어떻게 되지?

두 주소 사이에 저장될 수 있는 객체의 수를 구함.

(ex.

q는 p + 3인 것으로 생각하여 계산함.
즉, 하나의 문자로 바꾼 뒤에 연산하면 됨.

+ %p

주소를 printf() 함수로 출력할 때는 %p 변환명세를 사용함.

주로 16진수로 주소 값을 출력함. (컴퓨터마다 다를 수 있음.)

2. 포인터의 활용

1. 참조에 의한 호출 구현하기

포인터로 주소를 다루면 다른 함수, 블록의 변수를 직접 사용할 수 있음. (참조에 의한 호출 구현 가능.)

return은 하나의 값 만을 전달할 수 있지만, 이 기능을 이용하면 여러 개의 값을 전달할 수 있음.

메모 포함[이16]: 하나의 값 만을 가진다는 것은 '수학적 함수'의 정의로 이해해 볼 수 있다.

1) 방법

1. 변수의 주소를 해당 변수를 다룰 위치로 전달. (함수 사용 시에는 매개변수 등으로 전달.)
2. * 연산자(역참조 연산자)를 사용해서 해당 변수를 직접 사용.

2. 두 변수의 값을 바꾸는 함수

1) 기본 메커니즘

1. 새로운 변수 하나를 정의. (주로 이 변수의 이름은 tmp로 함.)
2. 해당 변수를 값을 잠시 보관해 둘 공간으로 사용하여 두 변수의 값을 바꿈.

2) 포인터 이용

포인터를 이용해서 두 변수의 값을 바꾸는 함수를 간단하게 만들 수 있음.
참조에 의한 호출을 구현한 것처럼, 두 변수의 주소를 전달받아서 교환하는 방식임.

+ scanf() 함수의 인자에서 & 연산자를 사용하는 이유

scanf() 함수는 한 번에 여러 개의 값들을 처리하는 함수이기 때문에 return 을 사용하는 방식으로는 정보를 모두 처리할 수 없음. 그래서 포인터를 사용하여 해당 주소에 직접 값을 저장함.

+ 함수 호출 시 인자의 포인터

포인터가 정상적인 값을 가지고 있는지 확인하기 위해 if 문을 사용해서 해당 포인터의 값이 NULL 인지를 검토하는 것이 좋음.

3. 포인터와 배열

1. 배열 이름

배열 이름은 해당하는 배열의 시작 주소를 가리키는 상수 포인터임.
-> 배열 이름에 값을 대입하거나 변경하려고 하면 오류가 발생함.

각괄호를 생략했다면 배열 이름임.

배열 이름에 해당하는 배열은 각괄호에 생략으로 이해할 수 있음.
배열 이름의 값은 해당하는 배열의 시작 주소임.
배열 이름의 형은 해당하는 배열의 일차원 원소를 가리킬 수 있는 포인터 형임.

메모 포함[이17]: 값이 고정된 포인터.
일반 포인터와 다른 점은 값이 고정되었다는 점이다.

메모 포함[이18]: ex.
원소의 형이 int인 경우, * int.
원소의 형이 int [5]인 경우, int (*)[5]. (이차원.)

메모 포함[이19]: 원소 만큼의 크기를 가져야 주소 연산으로 다음 원소를 찾을 수 있다.

+ 함수와 배열의 이름

함수 인자로 배열 이름을 작성하는 것은 배열의 시작 주소를 전달하는 것임.
즉, 해당 함수의 매개변수로 작성한 배열에는 인자로 작성한 시작 주소가 값으로 들어감.

함수의 인자로 배열의 시작 주소가 아닌 주소를 전달해도 됨.
이 경우, 해당 주소가 시작 주소인 것처럼 취급됨.

함수의 매개변수로 작성하는 배열은 가독성을 높이기 위한 것일 뿐, 정적 메모리 할당이 일어나거나 하는 것이 아님. 즉, 어떤 값을 쓰더라도 전혀 상관없음.

주소를 전달하기 때문에 함수 호출이 끝난 후에도 배열에 값이 반영되는 것.

+ 배열의 메모리 크기 구하기

sizeof 연산자에 배열 이름을 넣으면 해당 배열이 가지는 전체 메모리 크기가 나옴.

배열 이름이 가지는 메모리의 크기가 나오는 것이 아님.

2. 배열과 포인터의 표기법

*(ptr + a) <-> ptr[a]

배열 표현과 포인터 표현은 서로 전환할 수 있음.

1) 다차원 배열의 전환

괄호로 묶어서 전환함.

변환하는 과정에서도 *와 []의 연산 순서 잘 고려하기.

<변수>[i][j]
((<변수> + i) + j)
*(<변수>[i] + j)
*(<변수> + i)[j]

2) 표기법의 장단점

배열 표기법은 간단하고, 포인터 표기법은 더 자세한 기능을 사용할 수 있음.

포인터 표기법을 배열 표기법으로 바꾸어 표현하면 알아보기 더 쉬울 수 있음.
포인터 표기법을 사용하면 바이트 단위로 정보를 다룰 수 있음.

+ [](인덱스)

선언 시에는 그만큼의 메모리를 사용한다는 의미.
사용 시에는 시작 주소로부터 어느 정도 떨어진 메모리인지를 의미.

이렇게 생각하면 이해가 편함.

메모 포함[이20]: 배열의 메모리 구조와 주소를 생각해 보면 쉽게 이해할 수 있다.

메모 포함[이21]: 임의로 표기법을 바꾸어 사용하면 가독성을 떨어뜨릴 수 있기 때문에 좋은 것만은 아니다.

메모 포함[이22]: ex. malloc() 함수로 메모리를 할당 받아 사용할 때 포인터를 배열 표기법으로 나타내면 더 사용이 편리할 수 있다.

```
int *ptr = (int *)malloc(sizeof(int) * 10);
for(int i = 0; i < 10; i++)
{
  ptr[i] = i;
}
```

등등.

3. 컴파일 시 다차원 배열의 변환

메모 포함[이23]: 위에 필기한 것은 문법을 전환하여 사용하는 것에 대한 내용이었고, 여기서는 컴파일 시에 어떻게 전환되는지를 필기한다.

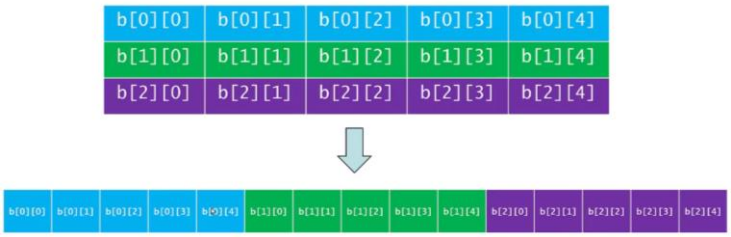
배열 표현 수식은 컴파일 시 모두 포인터 표현으로 변환됨.

1) 행우선 저장 방식

다차원 배열을 일차원 배열로 저장할 때는 낮은 차원부터 저장할 수도 있고, 높은 차원부터 저장할 수도 있음.

낮은 차원의 원소들부터 저장되는 것을 행우선 저장 방식이라고 함.

c에서는 행우선 저장 방식을 사용함.



2) 메모리 사상 함수 (변환 방식)

행우선 저장 방식을 통해 일차원 배열로 나열된 배열을 보면 주소를 특정한 방식으로 표현할 수 있음.

실제로 할당된 메모리 주소를 표현하는 방식임.

int array[a][b][c]; 인 경우, array[i][j][k] -> *(&array[0][0][0] + b * c * i + c * j + k) 임.

이렇게 변환하는 방식을 메모리 사상 함수라고 함.

차원이 높아질수록 연산의 양이 늘어남.

즉, 배열은 편리하지만 실행 시간이 길어질 수 있음. 포인터를 사용하면 더 빠름.

+ 배열의 최상위 차원 생략

함수 정의, 원형 선언 등에서 배열을 작성할 때 가장 높은 차원의 크기는 생략할 수 있음.

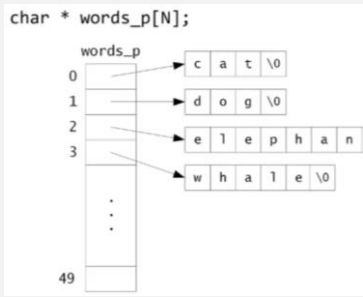
(배열 선언 시에는 메모리 할당을 위해 모든 차원을 명시해 줘야 함.)

매개변수로 작성하는 배열은 사실 포인터이고, 메모리 사상 함수에서 선언 시에 명시하는 가장 높은 차원의 크기(a)는 연산에 필요하지 않기 때문.

4. 배열과 포인터의 합체

변수 선언 시에 포인터와 배열 문법을 동시에 사용하는 방법.

[] 연산자가 * 연산자보다 우선순위가 높음.
괄호를 씌우지 않으면 []가 먼저 적용되어 배열이 되고, 괄호를 씌우면 *가 적용이 되어 포인터가 됨.



1) 포인터 배열(배열 기준)

<자료형> *<이름>[<크기>]

포인터 변수들로 이루어진 배열.
포인터 변수들이 배열 크기에 해당하는 개수만큼 선언되는 것.

포인터 배열의 자료형은 <자료형> *[<크기>]임.

일반 배열처럼 포인터를 사용할 수 있음.

배열은 포인터이므로, 포인터들의 집합인 포인터 배열은 일차원 배열들의 집합인 이차원 배열을 대체할 수 있음.
(ex, main 함수의 매개변수)

2) 배열 포인터(포인터 기준)

<자료형> (*<이름>)[<크기>]

지정한 배열 만큼을 포인트하는 포인터.
<자료형> (*)[<크기>] 만큼의 메모리 공간을 포인트함.

배열 포인터의 자료형은 <자료형> (*)[<크기>]임.

배열 포인터가 가리키는 배열 또한 포인터이기 때문에 배열 포인터는 포인터의 포인터임.

배열 포인터의 역참조는 해당 포인터가 포인트하는 배열의 이름으로 생각할 수 있음.
배열 이름은 해당 배열을 가리키고, 배열 포인터의 역참조 또한 해당 배열을 가리키기 때문.

주소 연산 시 한 번에 <자료형>[크기]의 크기만큼 주소가 증감함.

배열 포인터는 일차원 크기만큼의 배열을 포인트하기 때문에, 주소 연산이나 []를 사용해서 일차원 배열들의 집합인 이차원 배열을 대체할 수 있음.

메모 포함[이24]: 개념의 이해도가 조금 떨어지기는 하는데, 암기해서 사용하는 것이 더 나을 것 같다. 애초에 그리 많이 쓰이지도 않는 개념인 듯.

메모 포함[이25]: 원리보다는, 하나의 문법 약속으로 생각하는 것이 좋을 듯하다.

메모 포함[이26]: 이건 뇌피셜. 이해를 조금이나마 돕기 위해 정리했다.

5. 문자열과 포인터

컴파일러는 문자열 상수를 포인터로 취급함.

문자열 상수는 값으로 해당 문자열의 시작 주소를 가짐.
문자열 상수는 형으로 char * 형을 가짐. (해당 문자열의 원소를 가리킬 수 있는 자료형.)

문자열 상수는 포인터(주소 값)이기 때문에 인덱스([])를 사용할 수 있음.

1) 문자열 처리 시 배열과 포인터의 차이점

배열과 포인터는 대체로 비슷하게 사용되지만 약간의 차이점이 존재함.

포인터를 사용해서 문자열 상수를 다룰 때는 그 값을 변경할 수 없음.
문자열은 데이터 세그먼트에 저장되어 해당 주소를 가져와서 사용하기 때문에, 일반 변수와는 다르게 그 값을 변경할 수 없음.

메모 포함[이27]: ex. char *p = "abc"; 인 경우, p[2] = 'k'; 등으로 사용이 불가능하다.

문자열 상수를 배열에 넣어서 사용할 때는 그 값을 변경할 수 있음.
문자열 상수 자체의 값을 변경하는 것이 아니기 때문.

+ 포인터 사용 시 주의점

해당 공간이 실재로 존재하는지, 사용 가능한 공간인지를 고려해야 함.

할당 받지 않은 공간을 사용할 수는 없음.

(ex.
scanf() 함수로 값을 받을 때 포인터 변수를 사용하면 실행할 수 있음.
-> char *ptr; scanf("%s", ptr); 이렇게만 작성하면 실질적인 값을 저장할 공간이 없기 때문에 오류가 발생함.)

+ printf() 함수로 문자열 출력

printf() 함수에서 %s로 문자열을 출력할 때는 인자로 주소를 넣어 줘야 함.
printf() 함수로 문자열 출력 시, 작성한 주소부터 시작해서 NULL 문자가 나올 때까지 출력됨.

4. 동적 메모리 할당

1. 동적 메모리 할당

프로그래머가 원하는 시점에 원하는 크기만큼 사용할 수 있는 메모리 할당.
효율적인 메모리 사용을 위해서는 동적 메모리 할당이 필요함.

프로세스 중 스택 세그먼트의 힙(Heap)이라는 공간에 메모리를 할당함.
-> 프로그램이 끝날 때까지 자동으로 반납되지 않음.

메모 포함[이28]: 정적 메모리 할당보다 더 큰 공간을 사용할 수 있다.

2. 동적 메모리 할당 함수

이 함수들은 stdlib.h 헤더파일에 들어 있음.

1) calloc() (contiguous allocation)

인자에 지정한 크기만큼(바이트 단위) 힙 영역에 메모리를 할당하는 함수.

첫 번째 인자에는 원소의 개수를, 두 번째 인자에는 원소의 크기(바이트)를 작성함.

할당된 메모리의 시작 주소를 리턴함.
메모리를 할당 받지 못했다면 NULL을 리턴함.

할당한 메모리는 모두 0으로 초기화함.

자료형이 void * 형인 이유는 메모리의 사용 단위를 개발자가 결정할 수 있도록 하기 위함임.
리턴값을 포인터 변수에 대입할 때 사용 단위를 결정하여 바로 형변환 하는 것이 좋음.

메모 포함[이29]: ex.
int *ptr = (int *)malloc(100);
-> 사용 단위가 int(4바이트)인 경우
char *ptr = (char *)malloc(100);
-> 사용 단위가 char(1바이트)인 경우

함수 원형: void *calloc(size_t size, size_t el_size);
함수 사용 형식: void *p = calloc(10, 4);

+ size_t 자료형

size_t는 unsigned int와 동일한 자료형임.

2) malloc() (memory allocation)

인자에 지정한 크기만큼(바이트 단위) 힙 영역에 메모리를 할당하는 함수.

할당된 메모리의 시작 주소를 리턴함.
메모리를 할당 받지 못했다면 NULL을 리턴함.

할당한 메모리를 0으로 초기화하지 않음. -> 사용 전에 초기화가 필요함.

자료형이 void * 형인 이유는 메모리의 사용 단위를 개발자가 결정할 수 있도록 하기 위함임.
리턴값을 포인터 변수에 대입할 때 사용 단위를 결정하여 바로 형변환 하는 것이 좋음.

메모 포함[이30]: ex.
int *ptr = (int *)malloc(100);
-> 사용 단위가 int(4바이트)인 경우
char *ptr = (char *)malloc(100);
-> 사용 단위가 char(1바이트)인 경우

```
함수 원형: void *malloc(size_t size);
함수 사용 형식: void *p = malloc(100);
```

3) free() 함수

calloc(), malloc() 함수 사용 시 반환 받은 시작 주소를 인자로 넣으면 해당 함수가 할당한 메모리를 반납함.
이외의 값을 인자로 넣으면 오류가 발생함.

힙에 할당된 메모리는 프로그램이 끝날 때까지 자동으로 반납되지 않음. 직접 반납해 줘야 함.

```
함수 사용 형식: free(주소);
```

+ calloc(), malloc() 함수 사용 시 메모리 사용 단위를 더 쉽게 표현하는 방법.

sizeof 연산자를 사용해서 malloc() 함수의 인자를 더 보기 쉽게 표현할 수도 있음.

사용할 자료형의 크기를 sizeof 연산자로 나타내고, 사용할 총 묶음의 개수를 곱해주면 됨.

```
malloc(sizeof(<자료형>) * <묶음의 개수>)
calloc(<묶음의 개수>, sizeof(<자료형>))
```

+ calloc(), malloc() 함수 사용 시 주의할 점들.

1. free() 함수를 실수로 작성하지 않을 경우, 메모리가 엄청나게 낭비될 수 있음.
(함수가 여러 번 호출되는 등의 상황에서.)
-> malloc() 함수를 사용하는 경우, free() 함수를 함께 우선 작성해 주는 것이 좋음.

메모 포함[이31]: 힙에 할당된 메모리는 프로그램이 끝날 때까지 자동으로 반납되지 않기 때문에, free() 함수를 사용하지 않은 채 함수 호출을 반복하면 매번 다른 메모리를 계속 할당 받게 된다.

2. malloc() 함수가 반환한 시작 주소를 잃어버릴 경우 free() 함수를 사용할 수 없게 될 수 없음.

3. 메모리 할당에 실패하여 NULL 을 리턴하는 경우를 예외처리 해줘야 함.

3. 동적 메모리 할당의 활용

1) 동적 메모리 할당을 일차원 배열처럼 사용하기

포인터 변수는 자신의 자료형 크기 만큼의 메모리만 가리킬 수 있으므로, 동적 메모리 할당으로 할당 받은 전체 메모리의 크기가 자료형보다 크다면, 한 번에 사용하지 못하고 남는 부분들이 생김.
이때 주소 연산을 사용해서 포인터 변수에 숫자를 더하거나 빼면 나머지 메모리 공간을 차곡차곡 사용할 수 있음.

이 성질을 사용하면 *(ptr + 0), *(ptr + 1), *(ptr + 2) 또는 ptr[0], ptr[1], ptr[2] 등으로 할당 받은 메모리 공간을 배열처럼 사용할 수 있음.

메모 포함[이32]: ptr++ 등을 사용해서 포인터 변수의 값을 바꿔버리면 처음에 malloc() 함수가 반환한 시작 주소를 잃어버릴 수 있기 때문에 이렇게 사용하는 것이 좋다.

2) 동적 메모리 할당을 이차원 배열처럼 사용하기

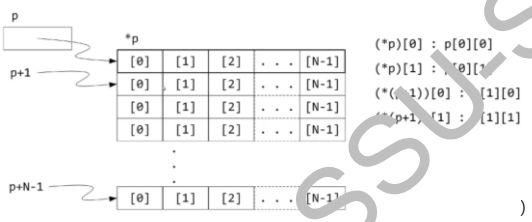
배열 포인터를 선언하여 할당 받은 메모리의 시작 주소를 배정하면, 이차원 배열처럼 사용이 가능함.
-> 배열 포인터의 원리를 사용하는 것이고, 동적 메모리 할당으로는 메모리 공간을 제공할 뿐임.

int (*ptr)[N];
ptr = (int (*)[N])calloc(N * M, sizeof(int));
-> 이렇게 작성하면 ptr 을 ptr[N][M] 이차원 배열처럼 사용할 수 있음.
-> *ptr, *(ptr + 1) . . . *(ptr + (N - 1))은 배열 이름임.

(ex.

프로그램 7.6

```
6    int (*p)[N], *q;
8    p = (int (*)[N])calloc(N * N, sizeof(int));
```



)

3) 할당 크기 변수로 지정하기

프로그램 실행 중에 크기를 결정하여 메모리를 할당 받을 수 있음.

배열은 정적 메모리 할당 방식으로 메모리가 할당되기 때문에 선언 시에 그 크기를 고정해 줘야 함.
반면에 malloc()은 함수이기 때문에 인자에 변수가 들어갈 수 있음.

5. 함수 포인터

1. 함수 포인터

특정 함수를 구성하는 첫 번째 명령을 가리키는 포인터.

프로세스는 코드 세그먼트, 데이터 세그먼트, 스택 세그먼트로 구성됨.
이 말은, 프로세스 상의 코드들도 메모리 상에서 주소를 갖는다는 의미임.
포인터 개념을 사용해서 특정 명령문의 주소로 이동하여 그 명령문을 실행할 수 있음.
단, 데이터의 구조를 망가뜨리지 않기 위해 함수 단위로만 주소를 사용해야 함.

함수 포인터는 일반 포인터와 동일한 메모리 크기를 가짐. (시스템에 따라 4 바이트/8 바이트 크기를 가짐.)

1) 선언

함수 포인터는 함수 원형을 사용해서 선언함.

<자료형> (*<식별자>)(<목록>);

자료형으로는 해당 함수 원형의 자료형을 작성함.
식별자로는 함수 포인터의 식별자를 작성함.
목록으로는 해당 함수 원형의 매개변수 목록을 작성함. (원형 선언과 동일한 방식으로 작성하면 됨.)

2) 배정

함수 포인터에는 함수의 식별자를 배정함.

<식별자> = &<식별자>;

함수 식별자 앞에 & 연산자를 붙여 함수 주소를 구할 수 있음.
& 연산자는 생략 가능.

3) 사용

역참조 연산자, 함수 포인터, 인자 목록을 명시하여 해당 함수를 호출할 수 있음.

역참조 연산자는 생략할 수 있음.

(*<식별자>)(<목록>)

식별자로는 함수 포인터의 식별자를 작성함.
목록으로는 인자 목록을 작성함.

+ 함수 포인터와 함수 원형 선언

함수 호출과 동일하게, 함수 포인터로 어떤 함수를 가리키기 전에 해당 함수의 정보가 확인되어야 함.
함수 정의나 함수 원형 선언이 코드 앞쪽에 존재해야 함.

메모 포함[이33]: 함수 포인터의 선언에서 *ptr은 모든 경우에서 괄호로 묶는다.
(이 수업의 내용에서는.)

메모 포함[이34]: 이해하는 것이라기보다는 그냥 이 문법의 사용 방법이다.

메모 포함[이35]: 함수 첫 번째 명령을 가리키는 것이 함수 포인터지만, 함수들끼리 명령이 겹칠 수 있기 때문에 함수 식별자를 사용한다.
(ex.
int a; 같은 것으로는 함수들을 구분하기 힘들 수 있다.)

2. 함수 포인터를 함수의 매개변수로 사용하기

함수 포인터를 함수의 매개변수로 사용하면 해당 함수의 수정을 용이하게 할 수 있음.
-> 라이브러리나 운영체제 등을 위한 함수에 주로 사용함.

1) 사용 방법

1. 함수 포인터 변수를 함수의 매개변수로 작성한다.
2. 해당 함수 포인터를 사용해 코드를 구성한다,
3. 함수 호출 시에 포인터 변수에 해당하는 인자에 함수의 식별자를 지정한다.
4. 지정한 함수가 함수 포인터 위치에 들어간다.

메모 포함[이36]: 해당 함수 내에서 사용하는 함수를 유동적으로 바꾸어 사용할 수 있다. 이 개념을 사용하지 않고 제작한 함수는 내부에서 사용하는 함수를 변경할 수 없다.

2) 퀵 정렬 함수 : qsort()

함수 포인터를 사용해서 정렬하는 함수.
다양한 형의 배열을 퀵 정렬로 정렬함.
stdlib.h 에 들어 있음.

```
void qsort(<배열이름>, <원소개수>, <메모리크기>, <함수식별자>);
```

첫 번째 인자 : 정렬할 배열의 이름.
두 번째 인자 : 정렬할 배열 원소의 개수. (보통 배열의 크기.)
세 번째 인자 : 각 원소의 메모리 크기. -> 배열 전체 크기 아닌 것 유의하기.
네 번째 인자 : 특정 형식의 함수 포인터. (원형이 동일해야 함.)
해당 배열 이름(주소)부터 시작해서, 각 원소의 메모리 크기만큼 시작 주소를 이동해 가며, 원소 개수 만큼을 정렬하기 위한 연산을 진행하는 함수인 것으로 보임.

메모 포함[이37]: 배열 선언 시에 배열이름[<크기>]로 작성하는 것으로 순서를 외우자.

메모 포함[이38]: 실제로 배열 원소의 개수를 반으로 줄이고, 각 원소의 메모리 크기를 두 배로 절반만 정렬된다.

네 번째 인자의 함수 포인터로 들어갈 함수는 조건에 맞게 사용자가 직접 작성함.

함수 헤더는 함수 포인터의 것과 동일하게 작성해야 함.

```
int <식별자>(const void *p, const void *q))
```

p 와 q 는 해당 배열의 원소를 가리키는 포인터임.
void 형으로 받았기 때문에 * 연산자 사용 시에는 해당 배열의 자료형에 맞게 형변환해줘야 함.

리턴 값이 아래와 같도록 작성해야 함. (첫 번째 인자 : p, 두 번째 인자 : q)
정렬 순서에서 *p 가 *q 보다 앞에 있음. -> 음수 리턴.
*p 와 *q 의 정렬 순서가 같음. -> 0 리턴.
정렬 순서에서 *p 가 *q 보다 뒤에 있음. -> 양수 리턴.
이건 오름차순 기준이고, 내림차순으로 하려면 양/음수만 바꿔주면 됨.

메모 포함[이39]: 오름차순 -> p가 뒤에 있길 바란다.

3) qsort() 사용 예시

메모 포함[이40]: 암기하자.

1. 정수 정렬용

```
int compare_int(const void *p, const void *q)
{
   if (*(int *)p > *(int *)q)
      return 1;
   else if (*(int *)p < *(int *)q)
      return -1;
   return 0;
}

int array[N] = { 3, 6, 2, 0, 1, 10, 4, 8, 9, 7 };

qsort(array, N, sizeof(int), compare_int);
```

2. 문자열 정렬용 -> 그냥 외우는 게 좋을 듯.

```
   qsort(words_p, num_word, sizeof(char *), comp_voca);

int comp_voca(const void *p, const void *q)
{
   return strcmp(*(char **)p, *(char **)q);
}
```

3. 함수 포인터 배열 사용하기

함수 포인터 변수를 배열로 선언해서 여러 함수 식별자들을 저장하고, 경우에 따라 사용할 수 있음.

1) 선언

함수 포인터 변수를 배열로 선언한 것

<자료형> (*<식별자>[<크기>])(목록)

메모 포함[이41]: *보다 []가 먼저 적용되어 포인터 배열이다.

자료형으로는 해당 함수 원형의 자료형을 작성함.
식별자로는 함수 포인터의 식별자를 작성함.
크기로는 생성할 배열의 크기를 작성함.
목록으로는 해당 함수 원형의 매개변수 목록을 작성함. (원형 선언과 동일한 방식으로 작성하면 됨.)

2) 사용 예시

```
typedef double (*func)(double);
func x[3] = {sin, cos, tan};

for (int i = 0; i < 3; i++)
   printf("%s(%.3f) : %.3f\n", name[i], rad, x[i](rad));
```

2. 사용자 정의형

사용자가 직접 선언하여 사용하는 자료형.

배열, 구조체, 공용체, 열거형 등이 있음.

메모 포함[이42]: struct, union, enum

1. 구조체

메모 포함[이43]: 총 3가지 종류의 선언 방법이 있다.
1. 일반적인 선언.
2. 선언과 동시에 변수 선언. -> 태그 유무.
3. typedef 사용. -> 따로 빼서 작성, 태그 유무.

1. typedef (type define)

typedef <기존 자료형> <새 자료형>;

메모 포함[이44]: typedef A B -> A를 B로 정의한다.

배열, 포인터, 함수 포인터에서는 기존에 식별자가 들어가는 자리가 새 자료형을 작성하는 자리이다.

자료형의 이름을 사용자가 정한 것으로 바꾸어 사용할 수 있게 하는 문법.

함수 밖에 작성함. (주로 #include, #define 등의 바로 밑에 작성함.)

복잡한 자료형을 간단히 나타낼 수 있고, 공통된 자료형의 크기를 한 번에 일괄적으로 변경할 수 있음.

메모 포함[이45]: ex.
typedef int NEW[10];
int (*ptr)[10]; -> NEW *p로 쓸 수 있다.

1) 배열 재정의하기

배열의 자료형을 앞에, 배열의 크기를 새 자료형 뒤에 작성함.

(<자료형>[<크기>]가 기존 자료형임.)

typedef <자료형> <새 자료형>[<크기>];

메모 포함[이46]: ex.
typedef int NEW[10];

int main(void)
{
  NEW a; -> int a[10]; 와 동일하다.
  return 0;
}

2) 포인터 재정의하기

* 뒤에 새 자료형을 작성함.

typedef <자료형> *<새 자료형>;

이 경우 <자료형>형 포인터를 새 자료형으로 정의하는 것임.

(<자료형> *이 기존 자료형임.)

3) 함수 포인터 재정의하기

typedef <자료형> (*<새 자료형>)(<목록>);

자료형과 목록은 함수 원형과 동일하게 작성함.

(<자료형> (*)(<목록>)이 기존 자료형임.)

2. 구조체

다른 종류의 자료형을 가진 것들끼리 그룹화할 수 있도록 새로운 자료형을 생성하는 문법.
구조체를 이용하면 다양한 메모리 구조를 만들 수 있음.

배열은 편리하지만 같은 종류의 자료형을 가지지 않은 것들은 묶을 수 없기 때문에 구조체를 사용해야 함.

메모리에는 멤버를 선언한 순서대로 저장됨.

1) 구조체 선언

구조체를 나타내는 struct 키워드로 시작함.

struct-tag 는 구조체를 식별자하기 위한 이름. -> 필요에 따라 생략 가능.

구조체 내의 요소들을 멤버라고 함.
중괄호 안에는 구조체의 멤버들을 변수 선언하듯 나열하면 됨.
구조체 멤버의 이름은 해당 구조체 안에서만 겹치지 않으면 됨.

중괄호 맨 뒤에 세미콜론 붙이는 거 조심하기.

구조체 선언은 함수 내부에서도 할 수 있고, 외부에서도 할 수 있음.

```
struct <struct-tag>
{
        <자료형> <변수명>
        <자료형> <변수명>
        ...
};
```

메모 포함[이47]: 구조체 선언 외부에 있는 식별자와는 겹쳐도 상관없다.

해당 구조체를 사용하는 블럭 내에 동일한 이름의 변수가 있어도 된다.

2) 구조체 변수 선언

구조체를 사용해 만든 자료형을 선언문에서 사용할 때는 struct + < 이름 > 형태로 작성함.
struct 까지 붙이는 것 조심하기.

배열, 포인터 등도 자료형만 구조체이고, 선언 방식은 동일함.

```
struct <struct-tag> <변수명>;
```

+ 대문자 이름

#define 이나 typedef 를 사용할 때는 새로 정의했다는 것을 알리기 위해 주로 이름을 전부 대문자로 함.

+ typedef 와 define 의 차이점

typedef 는 자료형을 새로 정의하는 것이지만, define 은 단순 치환하는 것임.
(ex. typedef int* NEW;와 #define NEW int*는 NEW a, b;에서 다르게 적용됨.)

3. 구조체 멤버의 사용

<구조체 변수 이름>.<멤버명>

1) 구조체 변수가 일반 변수인 경우

구조체로 선언한 변수의 멤버를 사용하려면 . 으로 구분하여 요소의 이름을 명시해 줌.
숫자로 명시하는 배열과는 달리, 구조체는 각 멤버의 이름을 명시해 줘야 함.

구조체 변수의 멤버는 일반 변수처럼 사용할 수 있음.

구조체 변수의 멤버에 배열이 있는 경우에도 일반 배열과 동일하게 사용이 가능함.

메모 포함[이48]: 배열 이름으로도 사용이 가능하고, 인덱스로 각 요소들을 명시해서 사용할 수도 있다.

<구조체 변수 이름>[<요소>].<멤버명>

2) 구조체 변수가 배열인 경우

일반 변수인 경우와 동일한 원리임.
구조체 변수 이름에 작성하는 이름 뒤에 배열의 요소를 []를 이용해서 명시해 주면 됨.

3) 구조체 변수를 포인터로 나타내는 경우

구조체 변수의 주소를 포인터 변수에 저장하고 * 연산자를 사용해서 해당 변수를 사용하는 경우.

* 연산자가 . 연산자보다 우선순위가 낮아서 (*<포인터>).<멤버명>으로 작성해줘야 함.
괄호를 생략한 경우, 포인터 변수는 . 연산자를 바로 사용할 수 없어서 오류가 발생함.
매번 괄호로 묶는 것은 번거로우므로 c는 -> 연산자를 제공함. (괄호로 묶은 것과 동일한 기능을 함.)

구조체 변수를 포인터로 나타낼 때, 포인터 변수의 자료형은 가리키는 구조체 변수의 자료형과 동일해야 함.

(<포인터 변수 이름>).<요소 이름>
<포인터 변수 이름>-><요소 이름>

4) 구조체 변수의 배정

기본적으로 구조체 변수 자체에는 값을 배정할 수 없음.
값은 구조체 변수가 가지는 멤버에만 배정이 가능함.

같은 형을 가지는 구조체 변수끼리는 배정이 가능함.

+ 배열에 문자열 배정하기

선언 시에는 배열에 문자열을 바로 배정할 수 있음.

선언문이 아닌 경우에는 배열에 문자열을 바로 배정할 수 없음.
strcpy() 함수를 사용하거나 각각의 요소에 직접 배정해 줘야 함.

4. 구조체 선언 방식

구조체는 여러 가지 방식으로 선언이 가능함.

메모 포함[이49]: 구조체 태그 생략과 동시 변수 선언은 항상 가능한 것 같다. 다만 기능이 조금 달라지는 것 뿐.

1) 구조체 선언과 동시에 구조체 변수 선언하기

구조체 선언 중괄호 바로 뒤에 변수명을 작성하여 바로 구조체 변수를 선언할 수 있음.
이 경우에도 구조체는 선언된 것이기 때문에 해당 변수 외에 다른 변수의 형으로도 사용이 가능함.

구조체 태그를 생략하고 구조체 변수를 바로 선언할 수도 있음.
이 경우 해당 변수는 해당 구조체 형으로 사용이 가능하지만, 해당 구조체는 다른 변수의 형으로 사용이 불가능함.

구조체 태그를 생략하고 동시에 선언한 구조체 변수와, 이후 동일한 구조의 구조체를 선언하여 선언한 구조체 변수는 다른 형을 가지고 있는 것으로 취급됨. 즉, 이 경우 이 두 변수는 서로 대입이 불가능함.

```
struct <구조체 태그>
{
        <자료형> <변수명>
        <자료형> <변수명>
        . . .
} <변수명>;
```

```
struct
{
        <자료형> <변수명>
        <자료형> <변수명>
        . . .
} <변수명>;
```

2) typedef 로 구조체 선언하기

구조체를 사용해 만든 자료형으로 변수를 선언할 때 매번 struct 를 쓰는 것은 불편하므로, typedef 를 사용해 자료형을 간단히 표현할 수 있음.

typedef 를 따로 빼서 작성하지 않고, 구조체 앞뒤로 typedef 와 <이름>을 작성해서 사용할 수도 있음.
이 경우에도 구조체 태그는 생략할 수 있음.

```
struct <구조체 태그>
{
        <자료형> <변수명>
        <자료형> <변수명>
        . . .
};
typedef struct <구조체 태그> <이름>;
```

```
typedef struct <구조체 태그>
{
        <자료형> <변수명>
        <자료형> <변수명>
        . . .
} <이름>;
```

```
typedef struct
{
        <자료형> <변수명>
        <자료형> <변수명>
        . . .
} <이름>;
```

5. 구조체 변수의 초기화

struct <이름> <변수이름> = {"이준혁", 1, 2.2};

배열처럼 {}를 이용해서 선언 시에 초기화 할 수 있음.
구조체 요소들의 순서와 {} 안의 값들의 순서를 맞춰줘야 함.

초기화자의 개수가 멤버의 개수보다 적은 경우, 앞에서부터 초기화되고 해당되는 초기화자가 없는 부분은 초기화되지 않음.

.<멤버이름>=<초기화자>를 명시하여 특정 멤버를 초기화할 수도 있음.
특정 멤버를 지정하여 초기화하면 {}의 다음 값은 해당 멤버 바로 다음 멤버로 들어감.
배열에서 []로 인덱스를 명시하여 초기화하는 것과 동일한 방식임.

1) 구조체의 멤버로 구조체 변수가 들어간 경우의 초기화

구조체의 멤버로 구조체 변수가 작성된 경우의 초기화 시에는 {}로 초기화자를 묶어서 구조체 변수에 제공함.
배열의 것과 유사함.

2) 구조체 배열의 초기화

{}로 초기화자를 묶어서 각 원소에 제공함.
초기화할 멤버를 명시하거나 초기화할 원소를 명시하여 초기화할 수도 있음. (=<초기화자>로 대입.)

메모 포함[이50]: ex.
struct subject math = {"MATH", {"Jhun", 100}, "jun", 90}};

메모 포함[이51]: ex.
struct subject math = {"MATH", {"Jhun", 100}, "jun", 90}};

6. 복합 리터럴

(<자료형>){<초기화 목록>}

지정된 형의 이름 없는 객체를 생성하는 문법.
(자료형이 구조체일 필요는 없음.)

초기화 목록을 지정하여 초기화함.
구조체의 경우, .<멤버이름>을 명시하여 특정 멤버를 초기화할 수도 있음.
배열에서 []로 인덱스를 명시하여 초기화하는 것과 동일함.

1) 사용 방식

방법 1. 복합 리터럴로 만들어진 객체는 메모리 할당을 받음.
복합 리터럴로 만들어진 객체의 값과 형은 해당 객체의 주소와 형임.
& 연산자를 사용해서 포인터 변수에 주소를 배정하여 사용할 수 있음.

방법 2. 복합 리터럴로 만들어진 객체는 동일한 자료형을 가지는 변수에 배정할 수 있음.
즉, 복합 리터럴로 객체를 생성하여 동일한 자료형의 구조체 변수에 값을 간단하게 배정할 수 있음.

메모 포함[이52]: ex.
typedef struct
{
  char a[5];
  int b;
} TEST;

TEST *ptr = &(TEST){ "jun", 20 }

메모 포함[이53]: ex.
typedef struct
{
  char a[5];
  int b;
} TEST;

TEST var = (TEST){ "jun", 20 }

-> 번거롭게 일일이 대입하지 않아도 된다.

7. 구조체 멤버로 구조체 작성하기

구조체 선언 시에 멤버로 구조체 변수가 올 수 있음.
이때 구조체 변수의 구조체는 미리 선언되어 있어야 함.

.<멤버명>을 여러 번 사용하여 구조체 변수의 멤버의 멤버를 사용할 수 있음.

8. 구조체 멤버 정렬

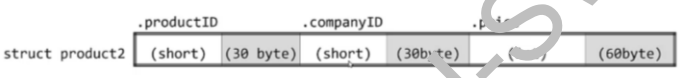
구조체의 멤버도 개발자가 직접 정렬하여 특정 주소에 저장되게 할 수 있음.

정렬하고자 하는 멤버 앞에 _Alignas()를 사용하여 정렬할 수 있음.
일반 변수와 동일하게, 선언문에서 자료형 앞에 명시함.

지정한 정렬 방식에 따라서 구조체의 전체 크기가 달라질 수 있음.

(ex.

```
struct product2{
    short productID;
    _Alignas (32) short companyID;
    _Alignas (64) int price;
};
```



)

9. 구조체와 함수

구조체 변수도 함수의 매개변수로 사용될 수 있고, 리턴될 수 있음.

배열과는 달리 값의 의한 호출로 전달됨.
배열처럼 주소를 전달하는 것이 아니라, 일반 변수처럼 해당 구조체 변수의 값으로 초기화함.

구조체의 멤버 중 배열이 있어도 해당 배열은 주소가 아닌 값이 전달됨.

구조체의 값을 전달받아 사용하고, 동일한 구조체로 리턴하여 변경한 값을 그대로 다시 전달할 수 있음.
구조체의 멤버 수가 많거나 큰 배열을 사용하는 경우 이 방식은 비효율적이기 때문에, 대부분의 프로그램에서는 함수의 인자로 구조체의 주소를 전달함.

메모 포함[이54]: 일반 변수에서는 그렇지 않은데, 구조체의 정렬 시에는 맨 뒤의 메모리 공간까지 그 크기로 친다.

메모 포함[이55]: 생각해보면 당연한 것이, 배열은 각 원소의 크기가 일정하기 때문에 주소와 자료형만 가지고도 값들을 편리하게 사용할 수 있다. 하지만 구조체는 각 멤버의 자료형이 일정하지 않기 때문에 주소로는 값을 편리하게 사용하기 어렵다.

10. 플렉시블 구조체

플렉시블 배열 멤버를 가지는 구조체.

구조체 멤버의 메모리 크기를 프로그램 내에서 지정할 수 있게 하는 문법임.

```
struct <구조체 태그>
{
        char a;
        char b[];
};
```

1) 플렉시블 배열 멤버

두 개 이상의 멤버를 갖는 구조체에서 크기가 정의되지 않은 마지막 배열 멤버.

2) 특징

플렉시블 구조체 변수를 선언하면 플렉시블 배열 멤버는 메모리 할당을 받지 않음.
이 경우 메모리 할당이 되지 않아 플렉시블 배열 멤버를 사용할 수 없음.

플렉시블 구조체의 크기는 항상 플렉시블 배열 멤버를 제외한 멤버들의 총 메모리 크기임.
동적 메모리 할당으로 메모리를 할당해도 구조체(자료형)의 크기는 동일함.

3) 플렉시블 구조체와 동적 메모리 할당

플렉시블 구조체는 동적 메모리 할당으로 메모리를 할당하여 사용함.
구조체 변수를 사용하기보다는, 해당 구조체를 가리킬 수 있는 포인터에 동적 메모리 할당으로 얻은 주소를 대입해서 사용함.

플렉시블 배열 멤버는 이름 그대로 배열이기 때문에, 해당 멤버에만 동적메모리할당으로 메모리를 할당할 수 없음.

기존 플렉시블 구조체의 크기와 플렉시블 배열 멤버가 가지는 크기를 더한 만큼을 할당함.

```
<자료형> *ptr = (<자료형> *)malloc(sizeof(<자료형>) + <플렉시블 배열 멤버가 가질 크기>);
```


11. 구조체의 메모리 크기

컴파일러는 구조체를 구성하는 멤버들 중 가장 크기가 큰 것을 기준으로 하고 그것의 배수로 멤버들의 메모리 공간을 하나씩 맞춤.
이에 따라 구조체의 메모리 크기는 멤버들 중 가장 크기가 큰 것의 크기의 배수가 되고, 멤버들의 위치에 따라 그 크기가 달라질 수 있음.

비트 필드를 사용하는 경우, 작성한 자료형의 크기를 기준으로 함.

메모 포함[이56]: ex.
struct test
{
 char a;
 int b;
 char c;
};
인 경우, sizeof(struct test)는 12이다.

메모 포함[이57]: 확실하진 않다. 테스트 해보니까 그런 것 같기는 하다.

2. 공용체

1. 공용체

구조체와 거의 유사하지만 각 멤버들이 같은 메모리 공간을 공유하는 사용자 정의 자료형.

동일한 종류의 값을 상황에 따라 다른 자료형으로 저장할 때 사용함.

공용체의 메모리 크기는 공용체의 멤버들 중 가장 큰 메모리 크기를 가지는 것의 크기임.

구조체와 동일한 문법 형식을 가졌음.

1) 공용체 선언

공용체를 나타내는 union 키워드로 시작함.

union-tag 는 공용체를 식별자하기 위한 이름. -> 필요에 따라 생략 가능.

중괄호 안에는 구조체의 멤버들을 변수 선언하듯 나열하면 됨.
공용체 내의 요소들을 멤버라고 함.

중괄호 맨 뒤에 세미콜론 붙이는 거 조심하기.

```
union <union-tag>
{
    <자료형> <변수명>
    <자료형> <변수명>
    . . .
};
```

2) 공용체 변수 선언

구조체와 동일한 방식으로 선언함.

```
union <union-tag> <변수명>;
```

3) 공용체 멤버의 사용

공용체로 선언한 변수의 멤버를 사용하려면 . 으로 구분하여 요소의 이름을 명시해 줌.

```
<변수명>.<멤버명>
```

공용체 변수의 멤버는 일반 변수처럼 사용할 수 있음.

각 멤버들의 형은 다를 수 있기 때문에 저장 시와 같은 멤버로 값을 사용해야 정확한 값을 사용할 수 있음.

메모 포함[이58]: 여긴 프로그래머의 꼼꼼함이 중요한 부분이다.

특정 값을 특정 멤버에 대입하면 해당 메모리 공간에 해당 멤버의 자료형으로 값이 들어가게 됨.
공용체 멤버들은 메모리 공간을 공유하므로 이때 다른 멤버들의 값에도 변동이 생김.

2. 공용체와 구조체

공용체는 주로 구조체의 멤버로 사용함.

구조체에 공용체 멤버를 사용할 때는 주로 구조체 멤버를 하나 더 사용하여 현재 어떤 공용체 멤버를 사용중인지 표시하는 용도로 사용함.

3. 익명 구조체와 익명 공용체

다른 구조체/공용체의 멤버로 구조체/공용체 태그 없이 선언된 구조체/공용체.

익명 구조체/공용체의 멤버는 그들이 속한 구조체/공용체의 멤버처럼 다뤄짐.
즉, 다른 멤버들과 동일한 방식으로 사용함. (. 연산자 하나를 사용하여 접근.)

1) 선언

속할 구조체/공용체 내부에 선언함.

익명 구조체/공용체는 구조체/공용체 태그를 가지지 않음.

익명 구조체/공용체 멤버의 이름은 그 익명 구조체/공용체가 속한 구조체/공용체의 멤버 이름과 겹치면 안됨.

익명 구조체/공용체 선언 시에도 끝에 ; 붙여야 되는 것 유의하기.

```
struct <struct-tag>
{
        <자료형> <변수명>
        union
        {
                <자료형> <변수명>
                <자료형> <변수명>
                . . .
        };
        <자료형> <변수명>
        . . .
};
```

메모 포함[이59]: ex.

```
struct test
{
  int a;
  union
  {
    int b;
    float c;
  }
}; 인 경우,
struct test var; 에서 var의 멤버에는 각각
var.a, var.b, var.c 와 같이 접근한다.
```

+ 공용체로 데이터를 비트 단위로 확인하기 (6 주차 실습)

공용체의 멤버들은 동일한 메모리 공간을 공유하기 때문에, 이를 이용해서 정수 자료형이 아닌 데이터들도 비트 단위로 확인할 수 있음.

하나의 멤버에 확인할 데이터를 저장하고, 정수 멤버로 그 데이터를 읽어서 비트 단위로 출력하는 방식임.

3. 열거형

1. 열거형

제한적이지만 숫자 대신 단어를 사용할 수 있게 해주는 문법.

기본적으로 프로그램에서는 숫자를 사용하지만, 숫자보다는 단어가 알아보기 쉬울 수 있기 때문에 열거형을 사용함.

enum 변수가 가지는 메모리 크기는 4 바이트로 고정임.

1) 열거형 선언

열거형을 나타내는 enum 키워드로 시작함.

enum-tag 는 열거형을 식별하기 위한 이름.

중괄호 안에는 열거자 목록을 작성함.

중괄호 맨 뒤에 세미콜론 붙이는 거 조심하기.

구조체와 동일한 방식으로 typedef 를 사용하여 선언할 수 있음.
구조체와 동일한 방식으로 선언과 동시에 변수 선언을 할 수 있음.

```
enum <enum-tag>
{
        <단어>,
        <단어>,
        . . .
};
```

2) 열거자 목록

선언 시 작성하는 열거자 목록에는 열거자들을 작성함.
열거자들은 이름만을 명시하고, 각 이름은 쉼표로 구분함. (초기화 목록과 동일한 형태를 가짐.)
열거자들의 이름은 해당 열거형의 유효범위 내에서 모두 유일해야 함.

열거자들을 정수처럼 사용됨.

값을 지정하지 않은 경우, default 값으로 첫 번째 원소부터 0, 1, 2, . . . 이 지정됨.
<단어>=<값>의 형식으로 값을 지정할 수 있음.
값을 지정한 경우, 지정된 열거자는 해당 값이 지정되고 그 이후 열거자는 앞 수로부터 1 커진 수가 지정됨.

메모 포함[이60]: 배열 초기화와 동일한 방식이다.

3) 열거자의 사용

열거형 정의가 되어 있다면 열거자들은 자신의 이름에 해당되는 정수 값을 가지게 되고, 열거자들을 작성하면 해당 정수 값으로 취급됨.

4) 열거형 변수

열거형 변수는 구조체와 동일한 방식으로 선언함.

```
enum <enum-tag> <변수명>;
```

열거형 변수는 일반 변수와 유사하게 사용할 수 있음.
열거형 변수에는 정수를 저장할 수 있음.
열거형 변수를 사용하는 이유는 자료형을 보기 쉽게 나타내기 위해서임. (특히 함수 정의 등에서.)

4. 자기 참조 구조체

1. 자기 참조 구조체

```
typedef struct test
{
        int data;
        struct test *ptr;
} TEST;
```

자신과 같은 구조체 형을 포인트하는 멤버를 가지는 구조체.
해당 구조체의 형을 형으로 갖는 포인터를 가진 구조체.

구조체들을 연쇄적으로 이어지게 할 수 있음.

메모리를 효율적으로 사용할 수 있게 하고, 가독성을 높이는 기능임.

1) 정의

구조체에 해당 자료형의 포인터를 멤버로 포함하여 정의함.

해당 구조체의 형을 형으로 하는 포인터의 자료형을 작성할 때는, 코드 앞쪽에 명시된 것을 사용해야 함.
즉, 구조체 태그를 명시해야 하고,여 struct <구조체 태그> 형태로 자료형을 작성해야 함.

자기 참조 구조체의 멤버는 해당 구조체를 가리킬 수 있는 포인터와 데이터를 저장할 저장공간으로 나뉨.

2) node

자기 참조 구조체 변수나 포인터로 생성한 각 메모리 공간을 node라고 함.

3) linked list

linked list는 node들의 집합임.
linked list에서 node들은 포인터로 서로 이어져 있음.

linked list는 여러 가지 방식으로 만들 수 있음.

메모 포함[이61]: c언어에서 가장 중요한 부분 중 하나.

메모 포함[이62]: ex.
typedef struct
{
 int data;
 TEST *ptr;
}TEST;
이렇게는 작성할 수 없다. 코드는 위에서 아래로 읽히기 때문에 TEST가 아직 읽히지 않았다.

메모 포함[이63]: ex. stack의 방식, Queue의 방식 등.

2. 동적 메모리 할당과 자기 참조 구조체

동적 메모리 할당과 자기 참조 구조체를 사용하여 프로그램 진행 도중 메모리를 관리할 수 있음.
효율적으로 메모리를 사용할 수 있음.

1) 방법

1. 구조체를 가리킬 수 있는 포인터를 생성한다.
2. 해당 포인터에 동적 메모리 할당으로 얻은 주소를 배정한다.
3. 해당 포인터에서 *나 -> 연산자를 사용하여 해당 메모리의 구조체를 가리키는 포인터에 접근한다.
4. 해당 포인터를 1 번의 포인터처럼 사용한다.

linked list 의 node 를 생각하면 간단함.

2) 주의점

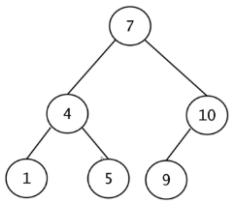
특정 메모리의 주소를 잃어버리면 gavage 가 생김.
주소를 잃어버리지 않아야 하고, 사용이 끝나면 free()로 반납해야 함..

gavage : 메모리를 차지하지만 접근할 수는 없는 공간.

+ 여러 개의 포인터 변수를 멤버로 갖는 자기 참조 구조체

자기 참조 구조체에 해당 구조체를 가리킬 수 있는 포인터 변수를 여러 개 넣어서 사용할 수도 있음.
이진 트리 자료구조에서 이 방식을 사용함.

이진 트리
각 노드가 최대 2 개의 자식 노드를 가지는 자료구조
왼쪽에 있는 자식 노드를 왼쪽 자식 노드, 오른쪽에 있는 자식 노드를 오른쪽 자식 노드라고 함.
루트 노드를 제외한 모든 노드는 부모 노드가 존재함.



(ex.

```
typedef struct node
{
        int data;           //데이터를 저장할 변수
        struct node *left;  //왼쪽 자식 노드를 가리키는 포인터.
        struct node *right; //오른쪽 자식 노드를 가리키는 포인터.
}NODE;
```

3. stack 구현하기 : 자기 참조 구조체 사용

추상 자료형 stack 을 자기 참조 구조체로 구현할 수 있음.
선형 linked list 로 구현함.

1) 데이터 집합 구현 방법

동적 메모리 할당과 자기 참조 구조체를 사용해서 linked list 를 생성함.

헤드 포인터(top) : 첫 번째 노드를 가리키는 포인터.

이 수업에서의 구현 방법. -> 헤드 포인터 쪽에 노드 추가.
1. node 구조체를 선언한다.
2. 헤드 포인터를 선언하고 NULL 을 배정한다.
3. 연산자를 사용하여 데이터를 관리한다.

2) 연산자 구현 방법

매개변수로 이차원 포인터를 사용하는 것은 헤드 포인터의 값이 변경되어야 하기 때문임.

1. push : 스택에 데이터 삽입.
-> 함수 원형 : void push(data d, NODE **top); (데이터, 헤드 포인터의 주소)
-> 새 노드를 헤드 포인터와 첫 번째 노드 사이에 끼워 넣고 데이터 저장.

2. pop : 스택의 최상위 데이터를 제거 후 해당 데이터를 리턴.
-> 함수 원형 : DATA pop(NODE **top); (헤드 포인터의 주소)
-> 첫 번째 노드를 제거하고 해당 데이터를 리턴.

3. top : 스택의 top 데이터를 리턴.
-> 함수 원형 : DATA top(NODE **top); (헤드 포인터의 주소)
-> 첫 번째 노드의 데이터 리턴.

4. empty : 스택이 비어 있는지 검사하여 리턴.
-> 함수 원형 : bool empty(NODE **top); (헤드 포인터의 주소)
-> NULL 여부를 체크하여 참 또는 거짓 리턴.

5. full : 스택이 꽉 찼는지 검사.
-> stack 의 크기가 고정된 경우에만 사용하는 기능이기 때문에 생략.

6. reset : 스택 초기화.
-> 함수 원형 : void reset(NODE **top); (헤드 포인터의 주소)
-> pop 연산자를 사용해서 모든 노드 제거.

메모 포함[이64]: 후입선출 방식.

메모 포함[이65]:
```
void push(int num, NODE **top)
{
    NODE *tmp_ptr = (NODE
*)malloc(sizeof(NODE));

    tmp_ptr->ptr = *top;

    *top = tmp_ptr;

    (*top)->data = num;

    return;
}
```

메모 포함[이66]:
```
int pop(NODE **top)
{
    int tmp_data;
    NODE *tmp_ptr;

    tmp_data = (*top)->data;
    tmp_ptr = (*top);

    (*top) = (*top)->ptr;

    free(tmp_ptr);

    return tmp_data;
}
```

4. Queue 구현하기 : 자기 참조 구조체 사용

메모 포함[이67]: 선입선출 방식.

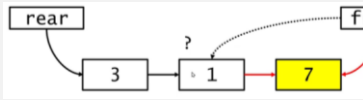
1) 데이터 집합 구현 방법

동적 메모리 할당과 자기 참조 구조체를 사용해서 linked list 를 생성함.

헤드 포인터(front) : 첫 번째 노드를 가리키는 포인터.
테일 포인터(rear) : 마지막 노드를 가리키는 포인터.
데이터 삽입은 rear 에서, 데이터 삭제는 front 에서 함.

데이터를 저장하는 각 노드 사이의 연결에는 두 가지 방식이 있음.
1. 포인터들이 rear->front 방향으로 가리키는 방식.
-> enqueue 시에는 문제가 없지만, dequeue 시에는 계산이 까다로워짐.
2. 포인터들이 front->rear 방향으로 가리키는 방식.
-> enqueue 와 dequeue 모두 간단함. Queue 구현 시에는 이 방식을 사용하는 것이 좋음.

메모 포함[이68]: dequeue시에 front가 가리키는 노드를 제거해야 하는데, 제거 이후 front가 가리켜야 할 노드의 주소를 알아내기 어렵다. 이를 알아내려면 rear부터 역참조 연산자를 사용해서 찾아내야 하기 때문에 상당히 번거롭다.



이 수업에서의 구현 방법.
1. 헤드 포인터, 테일 포인터를 선언하고 각각 NULL 로 초기화한다.
2. 연산자를 사용하여 데이터를 관리한다.

2) 연산자 구현 방법

매개변수로 이차원 포인터를 사용하는 것은 front 와 rear 의 값이 변경되어야 하기 때문임.

1. enqueue : 데이터 삽입.
-> 함수 원형 : void enqueue(int data, NODE **front, NODE **rear);
-> 노드 완성 후 끼워 넣는 것으로 이해.
-> 헤드 포인터가 NULL 인 경우와 NULL 이 아닌 경우로 나누어 처리.

NULL 인 경우
-> 새 노드 추가.
-> 새 노드의 저장공간에 데이터 배정.
-> 새 노드의 포인터에 NULL 배정.
-> front 와 rear 에 헤드 포인터 주소 배정.

NULL 이 아닌 경우
-> 새 노드 추가.
-> 새 노드의 저장공간에 데이터 배정.
-> 새 노드의 포인터에 NULL 배정.
-> linked list 에 해당 노드 끼워 넣기. (기존의 마지막 노드와 rear 값 수정.)

메모 포함[이69]:
```
void enqueue(int data, NODE **front, NODE **rear)
{
    NODE *tmp_ptr = (NODE *)malloc(sizeof(NODE));

    if((*front) == NULL) //노드가 없는 경우
    {
        (*front) = tmp_ptr;
        (*rear) = tmp_ptr;

        (*rear)->data = data;
        (*rear)->next_ptr = NULL;
    }
    else //노드가 있는 경우
    {
        (*rear)->next_ptr = tmp_ptr;
        (*rear) = tmp_ptr;

        (*rear)->data = data;
        (*rear)->next_ptr = NULL;
    }

    return;
}
```

2. dequeue : 데이터 삭제 후 해당 데이터 리턴.

-> 함수 원형 : int dequeue(NODE **front, NODE **rear);

-> 헤드 포인터가 NULL 인 경우와 NULL 이 아닌 경우로 나누어 처리.

NULL 인 경우 -> 0 리턴.

NULL 이 아닌 경우 -> 첫 번째 노드의 주소와, 첫 번째 노드에 저장된 데이터를 다른 변수에 배정.
-> 두 번째 노드의 주소를 front 에 배정.
-> 첫 번째 노드의 메모리를 free()로 반납.
-> 첫 번째 노드에 저장되었던 데이터를 리턴.

메모 포함[이70]:
```
int dequeue(NODE **front, NODE **rear)
{
    if((*front) == NULL) //노드가 없는 경우
    {
        return 0;
    }
    else //노드가 있는 경우
    {
        int tmp_data = (*front)->data;
        NODE *tmp_ptr = (*front);

        (*front) = (*front)->next_ptr;

        free(tmp_ptr);
        return tmp_data;
    }
}
```

+ 프로그램 제작의 순서

프로그램의 복잡도와는 상관없이, 프로그램 제작 시에는 반드시 다음과 같은 단계를 거치자.

1. 프로그램의 구조를 머릿속으로 그린다.
2. 각 요소별 구현 방법을 고안한다. -> 특히 구조체, 공용체 등 자료형 고안.
3. 코드를 작성한다.

3. 비트 단위 프로그래밍

1. 비트 단위 프로그래밍

1. 워드와 워드 경계

1) 워드

메모리 내용이 읽히는 단위

워드의 크기는 시스템에 따라 32 비트 또는 64 비트임.

2) 워드 경계

주소가 워드의 크기로 딱 나누어 떨어지는 부분.
메모리는 워드 경계부터 워드만큼 씩 읽힘.

2. 비트

1) 데이터 표현 방식

비트 단위로 값을 다룰 때는 데이터를 16 진수로 표기하는 것이 좋음.

2) 최상위, 최하위 비트

최상위 비트 -> 비트열에서 제일 왼쪽 비트
-> 부호가 있는 비트열에서 최상위 비트는 부호 비트로 사용함. (0->양수, 1->음수.)
최하위 비트 -> 비트열에서 제일 오른쪽 비트.

+ 지정되지 않은 비트

어떤 수를 저장한 경우 남는 비트에는 0 이 들어감.

3. 비트 단위 연산자

논리 연산자	(단항) 비트단위 보수	~
	비트단위 논리곱	&
	비트단위 배타적 논리합	^
	비트단위 논리합	|
이동 연산자	왼쪽 이동	<<
	오른쪽 이동	>>

비트 단위 연산자는 정수 수식에만 사용할 수 있음.

시스템에 따라 결과가 달라질 수 있음.
-> 여기서는 1 바이트=8 비트, 32 비트 운영체제, 2 의 보수 정수 기준임.

비트 단위 연산자는 일반 연산자들과 동일하게 복합 배정 연산자로 사용할 수 있음.

비트단위 보수 연산자만 단항, 나머지는 이항 연산자임.

1) ~ 연산자 (비트 단위 보수 연산자)

피연산자의 각 비트의 0 을 1 로, 1 을 0 으로 바꾸는 연산자.

! 연산자와 다르다. 유의하기.

2) 비트단위 논리곱(&), 배타적 논리합(^), 논리합 연산자(|)

두 비트의 값을 비교하는 연산자들.

두 피연산자들의 서로 대응되는 위치의 비트끼리 연산됨.

& -> 둘 다 참이면 참(1).
^ -> 둘이 다르면 참(1).
| -> 둘 중 하나라도 참이면 참(1).

메모 포함[이71]: (ex.

a	b	a & b	a ^ b	a | b
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

)

3) 이동 연산자(<<, >>)

지정된 값의 비트열을 이동시키는 연산자.

전체 수식의 형은 승격된 왼쪽 피연산자의 형이다.

이동 연산자의 expression2 가 음수이거나 expression1 의 가능한 비트 수보다 크거나 같은 값을 갖는다면 그 수행 결과는 정의되지 않음.

1. << (왼쪽)
수식 1 을 수식 2 가 지정하는 수만큼 왼쪽으로 비트 단위로 이동시키는 연산자.
왼쪽 이동으로 발생하는 빈 비트 공간은 0 으로 채워짐.
n 비트 이동한 것은 해당 수식에 2^n 을 곱한 것과 같음.

<수식1> << <수식2>

메모 포함[이72]: 참고로 이동 연산자를 사용하면 곱하기 연산보다 계산이 더 빠르다.

2. >> (오른쪽)
수식 1 을 수식 2 가 지정하는 수만큼 오른쪽으로 비트 단위로 이동시키는 연산자.
수식 1 이 unsigned 형 인 경우 오른쪽 이동으로 발생하는 빈 비트 공간은 0 으로 채워짐.
수식 1 이 signed 형 인 경우 오른쪽 이동으로 발생하는 빈 비트 공간은 시스템에 따라 0 또는 1 이 채워짐..
-> 이동 연산자는 unsigned 에만 사용하는 것이 좋음.

<수식1> >> <수식2>

4. 마스킹 연산

&이나 | 를 사용하여 비트열의 특정 비트를 0 이나 1 로 만드는 것.
비트열의 특정 부분을 추출, 알아내는 것.

마스크 : 마스킹 연산을 위해 사용되는 상수나 변수.

1) 예시들

1. 양의 정수의 짝/홀 판별.

```
int i, mask = 1; //mask : 00000000 00000000 00000000 00000001
for(i = 0; i < 10; i++)
{
        printf("%d", i & mask); //i 가 홀수면 1, 짝수면 0 출력.
}
```

2. 특정 비트의 값 알아내기.

```
(v & 0x4) ? 1 : 0    -> 0x4 는 00000000 00000000 00000000 00000100 임.
                     -> v 의 3 번째 비트 값이 1 인 경우 1 이 리턴됨.
v & 255              -> 255 는 00000000 00000000 00000000 11111111 임.
                     -> v 의 하위 1 바이트의 비트 값이 리턴됨.
```

3. 비트 단위로 출력하기.

```
void bit_print(int a)
{
        int n = (sizeof(int) * 8}; //a 의 비트 수를 n 에 저장.
        unsigned mask = 1 << (n - 1);
        for(int i = 0; i < n; i++)
        {
                printf("%d", (a & mask) ? 1 : 0);
                mask = mask >> 1;
        }
}
```

+ 실습에 나온 배타적 논리합 연산자 ^의 성질

이진수 A, B 가 있을 때, (A ^ B) ^ B 연산의 결과값은 A 임.

5. 패킹과 언패킹

패킹 : 비트단위연산자를 사용해서 적은 바이트로 압축하는 것.
언패킹 : 패킹된 정보에서 원하는 정보를 추출하는 것.

패킹 예시.
밀어서 대입하거나 대입해서 밀거나 임.

- 직원 관리 프로그램에서 다음과 같은 정보를 다루어야 한다고 가정
  - 직원 ID : 6자리(10진수)
  - 직업형태 : 200가지
  - 성별
- 이 세가지 정보는 unsigned int에 패킹 가능
  - 직원 ID : 20비트
  - 직업형태 : 8비트
  - 성별 : 1비트
  - 총 : 29비트

함수 9.1

```
1 unsigned pack_employee_data(unsigned id, unsigned job, char gender)
2 {
3    unsigned  employee = 0;
4    employee |= id;           // employee = id
5    employee |= job << 20;
6    employee |= ((gender == 'm' || gender == 'M') ? 0 : 1) << 28;
7    return employee;
8 }
```

언패킹 예시.
마스크를 사용해서 해당 부분을 통째로 확인.

- 패킹된 정보는 사용하기 전에 언패킹해야 함
- 적절한 마스크 필요
- 직원 관리 프로그램에서
  - 직원 ID를 위한 마스크 : 20개의 1
  - 직업 형태를 위한 마스크 : 8개의 1
  - 성별을 위한 마스트 : 1개의 1

함수 9.2

```
1 void print_employee_data(unsigned employee)
2 {
3    unsigned id, job;
4    char gender;
5    id = employee & 0xFFFFF;
6    job = (employee >> 20) & 0xFF;
7    gender = (employee >> 28) & 1;
8    printf("ID : %u", id);
9    printf("작업 형태 : %u", job);
10   printf("성별 : %s\n", gender? "F" : "M");
11 }
```

6. 비트 필드

구조체나 공용체 멤버의 크기를 비트 단위로 지정하는 것.
사용 시에 일반 멤버와 동일하게 사용이 가능함.

비트 필드는 정수 자료형을 가지는 멤버에만 사용할 수 있음.

위에 작성한 대로 패킹과 언패킹을 하면 실수하기 쉽기 때문에 비트 필드를 사용함.

1) 폭

<자료형> <멤버이름> : <폭>;

폭은 해당 멤버에 할당될 메모리 비트 수를 의미함.
폭으로는 양의 정수 상수 수식만을 작성함.
폭으로 지정하는 비트 수는 해당 멤버의 자료형보다 커질 수 없음.

정렬을 위해 이름 없는 비트 필드나 폭이 0 인 비트 필드를 사용할 수 있음.
이름 없는 비트 필드 -> 해당 비트만큼은 건너 뛴다는 의미.
폭이 0 인 이름 없는 비트 필드 -> 해당 워드에는 더 이상 저장하지 않고 건너 뛴다는 의미.

메모 포함[이73]: (ex.

```
struct small_integers {
   unsigned   i1 : 7, i2 : 7, i3 : 7,
                  : 11,
              i4 : 7, i5 : 7, i6 : 7;
}
struct abc {
   unsigned    a : 1, : 0, b : 1, : 0, c : 1;
};
```

)

2) 컴파일러의 처리

컴파일러는 비트 필드를 사용한 멤버들을 최소의 워드로 패킹함.
컴파일러는 비트 필드가 워드 경계에 걸치지 않게 함

3) 비트 필드 사용 시 주의사항

1. int 형 비트 필드는 시스템에 따라 unsigned int 비트 필드로 다뤄질 수 있음.
-> 비트 필드에는 unsigned 만을 사용하는 것이 좋음

2. 비트 필드 배열은 사용이 불가능함.

3. 비트 필드에는 & 주소 연산자를 사용할 수 없음.

4. 포인터는 비트 필드를 직접 포인트할 수 없음.
-> &로 주소를 얻을 수 없기 때문에 포인트할 수 없는 것이 아니라, * 연산자로 접근 자체가 불가능함.
-> 대신 -> 연산자로는 포인트할 수 있음.

4. 전처리기와 매크로

1. 전처리기

1. 전처리기

굉장히 유용한 문법임.

1) 전처리 지시자

#<지시자 이름>으로 구성됨.
#과 지시자 이름 사이에는 공백이 있어도 됨.

```
#<지시자 이름>
```

지시자 이름으로는 define, undef, include, if 등이 있음.

전처리 지시자는 사용자가 만들 수 없음.

2) 전처리기 문장

전처리 지시자로 시작되는 문장.

전처리기 문장은 해당 문장이 위치한 지점부터 그 소스 프로그램의 끝까지 영향을 미침.
전처리기 문장의 영향은 프로그램 종료 전에 다른 지시자에 의해 제거될 수 있음.

전처리기 문장은 c 언어의 나머지 문법들과 독립적임.
소스 파일을 컴파일하면, 컴파일러는 우선 전처리기 문장을 호출하여 모두 c 언어 문법으로 변환하고, 그 이후에 실행 파일을 생성함.

2. 매크로

1. 매크로

#define 전처리 지시자로 만든 전처리기.

매크로에는 기호 상수, 문자열 대치, 매개변수를 갖는 매크로 등이 있음.

매크로 정의가 길어질 경우, 행의 끝에 ₩(역슬래시)를 삽입하면 다음 행에 이어서 작성할 수 있음.

1) 유용성

자주 쓰이는 값 대신에 사용하여 수정을 용이하게 함.
프로그램의 가독성을 높임.

2) 주의점

1. 매크로 정의 시에는 반드시 모든 것을 괄호로 묶어줘야 의미적 오류를 방지할 수 있음.
2. 매크로 정의 시, 마지막에 ; 을 붙이면 대체 시에 ; 까지 들어가게 됨.

2. 기호 상수/문자열 대치

#define 전처리 지시자는 컴파일 시에 A를 B로 그대로 대체함.
컴파일러는 A와 B에 어떤 것이 오든 반드시 대체함.
꼭 기호 상수나 문자열 대치가 아니어도, 자유롭게 사용이 가능함.

#define <A> <B>

1) 기호 상수

B로 숫자 등을 지정하여 상수처럼 사용하는 방식.
기호를 상수처럼 사용할 수 있도록 하는 매크로.

표준 헤더파일에도 기호 상수들이 선언되어 있음.

2) 문자열 대치

B로 문자열 등을 지정하여 c의 구문을 효율적으로 바꿔서 사용하는 방식.
단, 구문을 너무 많이 바꾸면 타인의 가독성을 떨어뜨릴 수 있음.

메모 포함[이74]: typedef는 뒤의 것이 사용되고,
매크로는 앞의 것이 사용된다.
순서 유의하기.

메모 포함[이75]: ex.
#define EOF (-1) //파일의 끝을 나타내는 기호 상수.
#define NULL 0

메모 포함[이76]: ex.
==와 =가 헷갈리므로, #define EQ ==로 선언하여 EQ를 사용한다.

3. 매개변수를 갖는 매크로

#define 전처리 지시자는 함수와 유사하게 사용할 수도 있음.

1) 선언

함수와 유사한 형태를 가짐.

```
#define <식별자>(<매개변수 목록>) <대체 목록>
```

식별자와 괄호 사이에는 공백이 있으면 안됨.

매개변수 목록에는 식별자만 작성함.

여러 개의 식별자를 작성할 수 있음.

매크로는 형을 확인하지 않음. -> 단순 대체되는 것이기 때문에 어떤 형이 와도 동일함.

메모 포함[이77]: ex. #define AREA(x, y) ((x) * (y))

2) 사용

<식별자>(<인자 목록>)을 작성하여 사용한 부분은 모두 대체 목록으로 대체되는데, 이때 인자로 작성한 것들은 대체 목록으로 그 값이 전달되어 대체됨. (함수와 유사한 방식임.)

인자 목록에 아무것도 작성하지 않을 수도 있음.

이 경우, 구문에 어긋나지만 않는다면 오류가 발생하지 않음.

3) 주의점

1. 매크로 사용 시 인자들이 여러 번 평가될 수 있음.

매크로는 단순 대체하기 때문에, 하나의 수식이 여러 군데에서 실행될 수 있음.

(ex. #define AREA(x) ((x) * (x)), AREA(++r)인 경우, ((++r) * (++r)) 로 대체되어 ++r 이 2 번 수행됨.)

4. 통합 자료형 매크로

통합 자료형 매크로는 _Generic()을 사용함.

```
_Generic(<수식>, <목록>)
```

1) _Generic()

지정된 수식의 자료형에 따라 목록에서 expression 을 선택하여 리턴함.
수식의 자료형에 해당되는 것이 없다면 지정한 default 값이 들어감.
단순히 리턴한다기보다는, 해당 위치에 expression 이 대체되는 것.
expression 으로는 함수 식별자, 문자열 등 대체할 것을 작성함,

목록에는 <자료형> : <expression>의 형태로 작성함.
default : <expression>으로 default 값을 지정할 수 있음.
각 요소들은 쉼표로 구분함.

2) 구현 방법

#define <식별자>(x) _Generic((x), <자료형> : <func1>, <자료형> : <func2>, . . .)(x)
등으로 작성함.

식별자를 명시하고 x 값을 넣으면, _Generic 에서 자료형에 따라 함수 식별자를 선택하고 맨 뒤 괄호 안에 x 값이 들어감. -> 해당 함수의 인자로 x 값이 들어가게 됨.

지정한 식별자를 함수처럼 사용할 수 있음.

5. # 연산자

```
#<값>
```

매크로 정의에서 형식 매개변수를 문자열화하는 단항 연산자.

(ex. #define string(a) #a 에서, string(James)는 "James"로 대체됨.)

+ 디버깅 방법

변수나 주소에 저장된 값이 무엇인지를 printf() 등으로 단계별로 출력하는 방법을 사용할 수 있음.

이 과정에서 # 연산자를 사용하면 굉장히 편리함.
(ex. #define PRINT4DEBUG(x) printf(#x " : %d₩n", x);)

6. ## 연산자

<수식1> # <수식2>

두 토큰을 결합하는 이항 연산자.

토큰을 결합한다는 것은 코드 상에서 하나로 붙인다는 것.
(ex. #define X(i) x ## i 인 경우, X(2)는 x2 로 대체됨.)

7. 매크로 결과 확인 방법

컴파일러는 실행 파일은 생성하지 않고, 전처리기의 수행 결과만을 확인하는 명령이 있음.

우분투 리눅스의 gcc 에서는 gcc 명령어에 -E 옵션을 붙이면 됨.

-E 옵션 사용 시 출력할 내용이 많다면 한 화면에 전부 담기지 않아 잘리는데, 이 경우 리다이렉션을 사용해서 편하게 내용을 확인할 수 있음.
저장 형식은 외울 필요 없음.

+ # 연산자와 문자열

매개변수를 갖는 매크로를 만들 때, 큰따옴표 안에 있는 것들은 인자에 작성한 것으로 대체되지 않음.

큰따옴표 안에 대체해 넣으려면, 큰따옴표 바로 옆에 #<문자>를 작성하면 됨.
두 문자열이 붙어 있으면 합쳐지는 것을 이용한 것.

(ex.
#define test(a) printf("a"#a"e"); 인 경우 test(bcd)는 printf("abcde");와 동일함.)

8. 미리 정의된 매크로

c 에는 미리 정의되어 있는 매크로들이 있음.

1) 특징

이 매크로들은 정의하지 않아도 사용할 수 있음.
이 매크로들은 프로그래머가 해지할 수 없음.

컴파일러에 따라 미리 정의된 매크로들이 추가적으로 존재하기도 함.

2) 미리 정의된 매크로들

이 매크로들은 밑줄 문자 2 개로 시작해서 밑줄 문자 2 개로 끝남.

__DATE__ : 컴파일 시의 날짜를 포함하는 문자열.
__FILE__ : 파일 이름을 포함하는 문자열.
__LINE__ : 현재 라인 번호를 나타내는 정수.
__STDC__ : 표준을 따르는 경우 1, 아닌 경우 0.
__TIME__ : 컴파일 시의 시간을 포함하는 문자열.
__STDC_HOSTED__ : 호스트 구현이면 1, 아니면 0.
__STDC_VERSION__ : long int 형의 연도와 월.
__func__ : 현재 수행중인 함수 이름을 포함하는 문자열.

3. 헤더 파일

1. 헤더 파일

매크로 정의, 함수 원형, 기호 상수 등이 정의되어 있는 파일.
확장자가 .h 임.

헤더 파일에는 1. 사용자가 제작한 것과 2. 컴파일러가 제공하는 것이 있음.
사용자가 제작한 헤더파일은 주로 소스 파일과 같은 디렉토리에 만듦.

메모 포함[이78]: 표준 헤더 파일.

1) 사용법

#include 전처리 지시자를 사용해서 헤더 파일을 가져올 수 있음.
#include 전처리 지시자 사용 방법은 두 가지가 있음.

#include <<헤더파일>>
#include "<헤더파일>"

1. 헤더파일을 <>로 묶는 경우.
-> 컴파일러가 제공하는 헤더 파일을 가져올 때 주로 사용.
-> 시스템이 정의한 디렉토리에서 헤더 파일을 탐색함.

2. 헤더파일을 ""로 묶는 경우.
-> 사용자가 제작한 헤더 파일을 가져올 때 주로 사용.
-> 우선 현재 디렉토리에서 헤더 파일을 탐색하고, 해당 파일이 존재하지 않으면 시스템이 정의한 디렉토리에서 헤더 파일을 탐색함.

2) 사용자 헤더파일 (우분투 리눅스 기준)

확장자가 .h 인 파일 안에 코드를 작성하기만 하면 됨.
별다른 형식이 있는 것은 아니고, 단순히 소스 코드 작성하는 것과 동일하게 작성해 주면 됨.

메모 포함[이79]: ex.

헤더 파일 10.1 (pi.h)

```
#include <stdio.h>
#define  PI  3.14
#define  CIRCUMFERENCE(x)  (2.0 * (x) * PI)
#define  AREA(x)           ((x) * (x) * PI)
#define  VOLUME(x)         (4.0 / 3.0 * (x) * (x) * (x) * PI)
#define  SURFACE_AREA(x)   (4.0 * (x) * (x) * PI)
```

헤더파일, 매크로, 함수 원형, 구조체 선언 등을 모아둠.
일반적으로 소스 코드의 내용을 헤더파일로 만들진 않음.

헤더파일은 주로 여러 개의 소스 파일로 이루어진 프로그램을 만들 때 제작함.
하나의 프로그램은 매크로 정의, 헤더 파일 include 등이 동일하므로 겹치는 내용들을 헤더 파일로 만들어 두는 것.

+ 우분투 리눅스 헤더파일

/usr/include 에 가보면 c 언어 헤더파일들이 있음.

4. 조건부 컴파일

1. 조건부 컴파일

프로그램의 특정 부분을 선택적으로 컴파일할 수 있게 하는 기능.

1) 필요성

다양한 종류의 운영체제나 컴퓨터에서 프로그램이 원활하게 작동할 수 있게 함.
이식성 높은 프로그램을 만들 수 있음.

디버깅 코드 등을 상황에 따라 사용할 수 있게 함.

2. 조건부 컴파일의 전처리 지시자들

#if, #ifdef, #else, #elif, #endif, #undef 전처리 지시자들을 사용함.

1) #undef

앞에서 정의한 매크로를 무효화하는 전처리 지시자.

```
#undef <이름>
```

무효화할 매크로의 이름을 명시함.

매크로를 다시 정의할 때 주로 사용함.

2) #ifdef / #ifndef / #endif / #else

해당 매크로의 존재 여부를 검사하는 전처리 지지자들.

#ifdef -> 해당 매크로가 존재하는 경우에 참인 전처리 지시자.
-> 참인 경우 다음 전처리 지시자 전까지의 명령들을 컴파일함.

#ifndef -> 해당 매크로가 존재하지 않는 경우에 참인 전처리 지시자.
-> 참인 경우 다음 전처리 지시자 전까지의 명령들을 컴파일함.

#else -> 거짓인 경우에 대한 전처리 지시자. (if 문의 else 와 동일한 기능.)
-> 거짓인 경우 다음 전처리 지시자 전까지의 명령들을 컴파일함.

#endif -> 맨 마지막에 작성하여 끝임을 알리는 전처리 지시자.

이 전처리 지시자들은 서로 헷갈릴 수 있기 때문에, 들여쓰기로 구분하는 것이 좋음.

```
#ifdef <이름>
<문장>
#else
<문장>
#endif
```

메모 포함[이80]: #의 앞에 들여쓰기 하거나, #의 뒤에 들여쓰기 하는 등.

3) #if / #elif / #else / #endif

c 언어의 if 문과 동일한 방식으로 사용하는 전처리 지시자들.

메모 포함[이81]: 표현법만 다르고 방식은 동일하다.

#if -> 수식이 참인 경우 다음 전처리 지시자 전까지의 명령들을 컴파일함.
#elif -> #if 가 거짓인 경우 중에 수식이 참인 경우 다음 전처리 지시자 전까지의 명령들을 컴파일함.
#else -> #if 나 #elif 가 거짓인 경우 다음 전처리 지시자 전까지의 명령을 컴파일함.
#endif -> 맨 마지막에 작성하여 끝임을 알리는 전처리 지시자.

#if, #elif 뒤에는 정수 수식이 옴.

```
#if <정수수식>
<문장>
#elif <정수수식>
<문장>
#else
<문장>
#endif
```

+ defined()

괄호 안에 명시된 식별자가 정의되어 있으면 1 을, 정의되어 있지 않으면 0 을 값으로 가지는 연산자.

#if defined(<이름>)은 #ifdef 와 동일한 기능을 가짐.

```
#if define(<이름>)
<문장>
#endif
```

3. 디버깅 코드

조건부 컴파일은 디버깅 코드에서 자주 사용됨.

(ex.
#define DEBUG 1 //디버깅 여부에 따라 1 이나 0 지정.
#if DEBUG
printf("debug : a = %d₩n", a);
#endif)

1) gcc 컴파일 옵션을 사용해서 디버깅하기

-D 옵션으로 #define 을 대체할 수 있음.

gcc -D <이름> <파일>은 #define <이름>과 같음. 해당 파일에 적용됨.
gcc -D <이름>=<값> <파일>은 #define <이름> <값>과 같음. 해당 파일에 적용됨.

값을 지정하지 않을 경우 해당 매크로는 1 을로 대체됨.

```
gcc -D <이름> <파일>
gcc -D <이름>=<값> <파일>
```

+ 매크로 중복 피하기

아래와 같은 방식으로 기호상수가 겹치지 않도록 할 수 있음.

#ifdef PI
#undef PI
#endif
#define PI 3.141592
이렇게 작성하면 PI 매크로의 존재여부와는 상관없이, 3.141592 로 대체되는 PI 매크로가 존재하게 됨.

5. 내용그래밍

1. 메모리 배치

1. 프로그램 실행 시 메모리 배치

1) 메모리 할당 과정

1. 메모리를 할당한다.
2. 프로그램을 디스크(저장공간)로부터 읽어서 할당된 메모리에 로딩한다.

이후 main 함수의 첫 번째 문장이 실행됨.

2) 프로그램의 메모리 공간

프로그램이 할당받은 메모리 공간은 해당 프로그램이 끝날 때까지 그 프로그램에 의해 사용됨.
프로그램 내에서 사용되는 메모리 공간은 이때 할당받은 공간 내에서 사용함.

+ 변수의 초기화

선언문에서 값을 배정하는 것을 초기화라고 함.

+ 정적변수

static 스토리지 클래스 키워드를 붙인 변수.

지역변수에 static 을 붙이면 자동으로 0 으로 초기화되기 때문에 0 으로 초기화하더라도 BSS 에 저장됨.
static 을 붙인 지역변수는 0 으로 초기화할 경우 BSS 에 저장된다는 의미인 듯.
초기화를 안 해도 값이 0 이니까.

메모 포함[이82]: 운영체제가 이 과정을 수행한다.

메모 포함[이83]: 해당 메모리 공간 내에서 변수 등에 메모리 공간을 할당한다.

메모 포함[이84]: 아마..?

메모 포함[이85]: 강의 다시 봐 보기.

3) 프로그램의 메모리 구조

프로그램에 할당된 메모리는 용도별로 분할되어 있음.
텍스트, 데이터, 힙, 스택 부분으로 나뉨.

스택과 힙은 구분되지 않는 하나의 공간을 사용함.

스택 -> 함수의 지역변수, 함수의 매개변수, 함수를 호출한 곳의 주소(리턴 주소)를 저장하는 공간.
-> 함수가 호출될 때 사용되는 공간.
-> 제한된 공간임. 너무 큰 크기의 지역변수는 실행되지 않을 수 있음.

힙 -> 동적 메모리 할당 시에 사용하는 공간.
-> 제한된 공간임. free() 함수를 적절히 사용해야 함.

텍스트 -> 프로그램의 실행 코드(문장)를 저장하는 공간.
-> 기계어로 된 코드가 로드됨.
-> 이곳에 있는 코드를 읽어서 프로그램을 실행함.

데이터 -> 전역변수, 정적변수를 저장함.
-> 초기화 여부에 따라서 전역변수/정적변수를 나누어 저장함.
-> 초기화되는 것은 그것의 내용을 저장하고, 초기화되지 않은 것은 그것의 이름과 크기를 저장함.
-> BSS : 초기화되지 않은 전역변수/정적변수를 저장하는 공간.

메모 포함[이86]: 프로그램의 실행은 pc 레지스터를 사용한다.

메모 포함[이87]: Block Started by symbol.

2. 메모리 스택

1) 함수 프레임

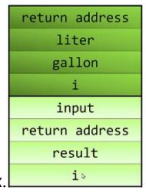
함수 호출 시에 해당 개별 함수의 지역변수, 매개변수, 리턴 주소로 이루어진 공간.

함수 호출 시에 함수 프레임이 스택에 삽입됨.
함수 프레임은 함수가 종료되면 삭제됨.
가장 나중에 호출된 함수가 가장 빠르게 종료되기 때문에, 함수 프레임은 후입선출 방식으로 삭제됨.

2) 메모리 스택의 대략적 구조

스택은 함수 프레임들로 구성되어 있음.

(ex.



main() 프레임
compute() 프레임

)

2. 대형 프로그램의 구성

1. 대형 프로그램의 구성

대형 프로그램은 주로 여러 개의 파일로 작성함.

주로 관련된 함수들을 하나의 파일에 저장함.

주로 한 프로그램을 이루는 파일들을 동일한 디렉토리에 저장함.

2. 다중 파일 프로그램의 컴파일

1) 전체 프로그램 컴파일

여러 개의 파일로 구성된 프로그램을 컴파일할 때에는 해당 파일들을 gcc의 인자로 모두 작성함.

-o <이름> 옵션을 사용해서 실행 파일의 이름을 직접 설정할 수도 있음.

```
gcc <file1> <file2> . . .
```

메모 포함[이88]: 즉, *.c 인 파일들만 인자로 작성할 수 있다.
파일 입출력으로 다루는 파일은 이곳에 작성하는 것이 아니다.

2) 파일 별 컴파일

프로그램을 여러 개의 파일로 작성한 경우, 각각의 파일들을 따로 컴파일할 수 있음. -> 디버깅 시 유용함.

-c 옵션을 사용해서 컴파일하여 해당 파일의 목적 파일을 생성할 수 있음.

```
gcc -c <file>
```

목적 파일은 .o 확장자를 가짐.
목적 파일들을 한꺼번에 컴파일하여 전체 프로그램 컴파일을 할 수 있음.
-> 이런 방식으로 할 경우, 수정한 파일만 다시 목적 파일만 생성하면 되기 때문에 유용함.

(ex.

```
gcc -c main.c                -> main.o 목적파일 생성.
gcc -c grade.c               -> grade.o 목적파일 생성.
gcc -o grade main.o grade.o  -> grade 실행파일 생성.
```

main.c 파일을 수정한 경우, gcc -c main.c 명령만 입력하면 다른 파일들은 컴파일하지 않고 main.c 만 컴파일하여 반영할 수 있음. 이후 gcc -o grade main.o grade.o 로 다시 컴파일이 가능함.)

3. 대형 프로그램의 사용자 헤더파일

메모 포함[이89]: 자세한 설명은 헤더파일 관련 필기 참고하기.

여러 파일로 구성된 대형 프로그램에서는 주로 사용자 헤더파일을 사용함.

사용자 헤더파일에는 헤더파일, 매크로, 함수 원형, 구조체 선언 등을 모아둠.

1) 헤더파일의 다중 include (include guard)

표준 라이브러리들의 헤더파일들과는 달리, 사용자 헤더파일은 다중 include 가 방지되어 있지 않음.
하나의 프로그램 내에서 동일한 사용자 헤더파일을 2 번 이상 include 한 경우 오류가 발생함.
표준 라이브러리들의 헤더파일들은 여러 번 include 해도 한 번만 적용됨.

매크로를 하나 사용하여 사용자 헤더파일의 다중 include 를 방지할 수 있음.

(ex.
#ifndef _TEST　　//_TEST 매크로가 존재하는 경우에만 아래의 명령 실행.
#define _TEST　　//_TEST 매크로 생성 -> 이후에 이 부분은 다시 실행되지 않음.
<명령>
#endif
)

3. 정적 외부 변수/함수

1. 외부변수

함수 밖에서 설명된 변수.

모든 함수에서 참조가 가능함. -> 함수 간 정보 전달에 유용함.

프로그램이 여러 파일로 이루어져 있어도 extern 으로 선언하여 외부 변수를 모든 파일에서 사용할 수 있음.
-> extern 으로 선언해야 다른 파일에서도 사용할 수 있음.
-> extern 을 이용한 전역변수 선언은 프로그래밍기초및실습의 필기를 참고하자.

2. 정적 외부변수

static 이 적용된 외부변수.

해당 외부변수가 정의된 파일 내에서만 해당 외부변수를 사용할 수 있음.

다른 파일에서의 데이터 접근을 제한하면서 해당 파일 내 함수 간 정보 전달에 유용함.
정보 수정/유출의 걱정 없이 더욱 안전하고 편리하게 프로그래밍할 수 있음.

3. 정적 외부함수

static 이 적용된 외부함수.
함수는 기본적으로 외부함수임. -> 원형 선언이 되어 있다면 모든 파일 또는 위치에서 호출할 수 있음.

정적 외부함수는 해당 함수가 정의된 파일 내에서만 해당 함수를 호출할 수 있음.

프로그램을 더 안전하게 만들어 줌.

4. 추상 자료형

1. 추상 자료형

처리해야 할 데이터 집합과 그 연산을 정의한 명세.
프로그램은 데이터의 처리가 주된 목적 중 때문에 데이터 처리에 대한 정의가 필요함.

데이터는 추상 자료형에서 정의된 연산자에 의해서만 다뤄짐.
-> 데이터가 보호됨.

추상 자료형에서는 해당 개념을 구현하는 방식은 고려하지 않음.
-> c 언어를 사용해 그 자료형을 구현하는 것은 고려된 것이 아님.

2. stack

후입선출 방식으로 데이터를 저장하는 자료구조. (LIFO, Last In First Out)

top : 현재 스택 내에서 가장 최근에 추가된 데이터.
후입선출이기 때문에 top 값만이 바뀜.

1) stack 의 연산자

push : 스택에 데이터 삽입.
pop : 스택의 최상위 데이터를 제거 후 해당 데이터를 리턴.
top : 스택의 top 데이터를 리턴.
empty : 스택이 비어 있는지 검사하여 리턴.
full : 스택이 꽉 찼는지 검사.
reset : 스택 초기화.

2) stack 구현하기 : 배열

구조체, 배열로 stack 구현함.

이 구조체의 배열에는 각 요소에 데이터 객체를 저장하고, top 에는 현재의 top 인덱스를 저장함.

(ex.

typedef struct　　//정수를 저장하는 스택을 구조체, 배열로 구현.

{

　　int data[100];

　　int top;

} STACK;

)

연산자 구현.

```
 1 #include "stack.h"
 2 void reset(stack *stk){
 4     stk -> top = EMPTY;
 5 }
 6 void push(char c, stack *stk) {
 8     stk -> top++;
 9     stk -> s[stk -> top] = c;
10 }
11 char pop(stack *stk) {
13     return (stk -> s[stk -> top--]);
14 }
15 char top(const stack *stk) {
17     return (stk -> s[stk -> top]);
18 }
19 bool empty(const stack *stk) {
21     return (stk -> top == EMPTY);
22 }
23 bool full(const stack *stk) {
25     return (stk -> top == FULL);
26 }
```

3) stack 구현하기 : 자기 참조 구조체(linked list)

자기 참조 구조체 부분에 필기.

3. Queue(큐)

선입선출 방식으로 데이터를 저장하는 자료구조. (FIFO, First In First Out)
먼저 저장한 것을 먼저 사용하는 자료구조.

rear : 끝.
front : 앞.
데이터 삽입은 rear 에서, 데이터 삭제는 front 에서 함.

1) Queue 의 연산자

enqueue : 데이터 삽입.
dequeue : 데이터 삭제 후 해당 데이터 리턴.
empty : 비었는지 검사.
full : 꽉 찼는지 검사.
reset : 큐 초기화.

2) Queue 구현하기 : 자기 참조 구조체(linked list)

자기 참조 구조체 부분에 필기.

6. 입출력 함수

1. 스트림

1. 스트림

다양한 입출력 장치를 일관성 있게 접근하게 해 주는 기능.
입출력 장치와 프로그램 사이에서 입출력 자료들을 중계하는 역할을 하는 것.

프로그램에서는 입출력을 위해 연관된 스트림을 생성하고 해당 스트림으로부터 읽고 씀.

1) 기본 스트림

프로그램에서 만들지 않아도 이미 생성되어 있는 기본 스트림들이 존재함.

표준 입력 장치(stdin, 1) : (default) 키보드. stdin 파일 포인터로 접근 가능.
표준 출력 장치(stdout, 2) : (default) 화면. stdout 파일 포인터로 접근 가능.
표준 오류 장치(stderr, 0) : (default) 화면. stderr 파일 포인터로 접근 가능.

표준 입출력 함수들은 기본 스트림을 사용함.

리다이렉션이나 파이프로 표준 입출력 장치를 변경할 수 있음.

메모 포함[이90]: 입출력 장치들의 접근 방식은 제각각이기 때문에 이 기능이 없다면 장치가 달라질 때마다 코드를 수정해야 할 것이다.

2. 입출력 재지정 (리다이렉션) (리눅스 명령어)

표준 입출력 장치를 재지정하는 기능.

1) > (출력 리다이렉션 방법 1)

출력되는 정보의 방향을 해당 파일로 리다이렉션함.

<command> <number>> <file>

정보를 저장할 파일이 이미 존재하면 덮어쓰기함. (원래의 정보를 지우고 해당 정보를 저장함)

지정한 이름의 파일이 없으면 해당 파일을 생성해서 수행함.

<number>에는 표준 입출력 장치의 파일 디스크립터를 명시함.
특정 번호를 명시하면 일시적으로 오른쪽에 오는 파일을 해당 번호로 취급하겠다는 의미.
아무것도 명시하지 않은 경우, 1 로 취급됨.
출력된 실행 결과를 리다이렉션하려면 1 번을, 출력된 오류 메시지를 리다이렉션하려면 2 를 작성함.

연속해서 리다이렉션할 수 있음.

2) >> (출력 리다이렉션 방법 2)

출력되는 정보의 방향을 해당 파일로 리다이렉션함.
출력되는 문자열을 해당 파일에 추가함.
>와는 달리 삭제하고 넣는 게 아니라 그냥 넣음.

<command> <number>>> <file>

지정한 이름의 파일이 없으면 해당 파일을 생성해서 수행함.

3) < (입력 리다이렉션 특수기호)

입력되는 정보를 해당 파일에서 가져오도록 리다이렉션함.
해당 file 의 데이터를 명령에 입력함.

<command> <number>< <file>

<는 0<를 생략하여 표현한 형태임.

+ 입출력을 한 번에 리다이렉션하기.

test < input_file > output_file 등으로 작성.

메모 포함[이91]: 표준 입출력 장치를 일시적으로 파일로 바꾸는 것.

3. 파이프(|)(리눅스 명령어)

왼쪽 명령의 수행 결과를 오른쪽 명령의 입력으로 전달하는 리눅스 쉘의 특수문자.

파일과 실행 파일을 파이프로 연결하여 파일의 데이터를 입력으로 받거나 파일에 출력을 쓸 수 있음.

여러 개의 파이프를 작성하여 연쇄적으로 사용할 수도 있음.

```
<cmd> | <cmd>
<cmd> | <cmd> | <cmd>
```

2. 입출력 함수들

1. getchar() / putchar()

1) getchar()

```
int getchar(void);
```

표준 입력 장치에서 문자 하나를 읽어서 리턴하는 함수.

읽을 문자가 존재하지 않으면 EOF 를 리턴.

2) putchar()

```
int putchar(int c);
```

표준 출력 장치로 문자 하나를 쓰는 함수.

매개변수로 출력할 문자의 값을 정수로 받음.

출력된 경우 출력한 내용을 리턴함.
출력되지 않은 경우 EOF 를 리턴.

+ EOF (End Of File)

파일의 끝을 알리는 기호 상수.
컴파일 시 -1 로 대체됨.

stdio.h 헤더파일에 들어 있음.

리눅스에서는 <ctrl> + d 를 입력하여 키보드로 EOF 를 입력할 수 있음.

+ 표준 입력 버퍼

운영체제가 제공하는, 사용자의 입력을 임시로 저장하는 메모리.
표준 입력 함수들은 표준 입력 버퍼를 사용함. -> 표준 입력 함수들의 작동 방식을 이해할 수 있음.

표준 입력 버퍼 사용 방식.
1. 표준 입력 버퍼에 내용이 남아 있다면 정보를 새로 입력 받지 않음.
1. 표준 입력 버퍼에 내용이 남아 있지 않다면 특정 키를 누를 때까지 정보를 입력 받아 표준 입력 버퍼에 저장함.
2. 표준 입력 버퍼의 내용을 사용함.

메모 포함[이92]: 한 번의 입력이 여러 함수에 들어가는 것이나, 하나의 표준 입력 함수에서 오류가 다른 표준 입력 함수에 영향을 끼치는 것 등.

2. printf()

표준 출력 장치로 문자열을 출력하는 함수.
첫 번째 인자의 제어 문자열을 출력하고, 출력한 문자의 개수를 리턴함.

1) 인자 작성법

첫번째 인자에는 출력할 문자열인 제어문자열이 들어감.
두번째 인자부터는 제어문자열 내의 변환명세에 대입될 값이나 변수, 수식이 들어감.
인자의 개수에는 제한이 없음.

```
int printf(const char * restrict format, . . . );
```

2) 변환명세

값을 어떤 형식으로 출력할 것인지 명시하는 키워드.
%로 시작하여 변환문자로 끝남.
출력할 데이터와 변환명세를 반드시 맞춰 줘야 함. 자동변환 없음.

변환 명세보다 들어갈 인자의 개수가 많다면 여분의 인자는 무시됨.
변환 명세보다 들어갈 인자의 개수가 적다면 시스템에 종속적인 결과가 발생함.

메모 포함[이93]: 경고 메시지가 나오고 이상한 값이 출력됨.
자료형이 달라도 큰 문제가 없는 경우에는 정상적인 값이 나오기도 함.

3) 변환 문자의 종류

c	문자
d, i	10진 정수
u	부호 없는 10진 정수
o	부호 없는 8진 정수
x, X	부호 없는 16진 정수, 예: 5dee, 5DEE
f, F	부동 소수점 표기법의 실수, 예: 7.123000
e, E	지수 표기법의 실수, 예: 7.123000e+00, 7.123000E+00
a, A (C99)	16진 지수 표기법의 실수, 예: 0x7.123p+20, 0X7.123P+20
g, G	e형식과 f 형식 중 짧은 것 또는 E형식과 F 형식 중 짧은 것
s	널로 끝나는 문자열
p	포인터 값의 16진 정수
n	출력되는 것 없음, 대응되는 정수형 포인터 인자에 현재까지 출력된 문자의 개수를 배정함
%	% 문자, 대응되는 인자 없음

%p 로 출력하면 16 진수로 주소 값이 출력되지만, 세부적인 형태는 시스템에 따라 다름.

대문자인 경우 알파벳이 대문자로 출력됨.

4) 예외적 상황들

출력할 데이터가 무한대인 경우 INF(infinity)를 출력함.
출력할 데이터가 숫자가 아닌 경우 NaN 을 출력함.

5) 변환 명세 옵션

%와 변환문자 사이에 flags, field-width, .precision, length-modifier 를 순서대로 작성해 옵션을 부여함.
각 옵션은 변환 명세의 종류에 따라 다르게 적용되거나 타 옵션 사용여부 등에 따라 적용되지 않을 수 있음.

1. flags -> flag. 출력 형식 조정.
-> 오른쪽 정렬이 default 값임.
-> +와 공백이 같이 사용되면 공백 flag 는 무시됨.
-> %x 에서 0 을 지정하면 추가적 0 은 0x 뒤에 옴.

플래그	의미
-	필드에서 좌측 정렬로 출력
+	숫자 앞에 +나 -를 항상 붙임 출력
공백	양수 앞에 공백을 붙여 출력
0	숫자를 우측 정렬로 출력할 때 남는 공간을 0으로 채워서 출력
#	8진수는 앞에 0, 16진수는 앞에 0x, 실수는 소수점을 출력

2. field-width -> 필드 크기. 변환 명세 자리에 출력할 전체 칸 수를 지정.
-> 필드 : printf()에서 인자가 출력되는 공간.
-> 양의 정수 또는 *을 작성함.
-> *는 인자로 필드 크기를 지정할 수 있음.

3. .precision -> 정밀도. 실수 출력 시 소수점 몇 번째 자리까지 출력할 것인지 지정.
-> 음이 아닌 정수 또는 *을 작성함.
-> *는 인자로 정밀도를 지정할 수 있음.
-> 변환문자의 종류에 따라 다르게 적용됨.
d, i, o, u, x, X : 출력될 숫자의 최소 자릿수.
a, A, e, E, f, F : 출력될 숫자의 소수점 이하의 자릿수.
g, G : 최대 유효 숫자.
s : 문자열로부터 출력될 문자의 최대 개수.

4. length-modifier -> 형변환자. 인자 출력 시 자료형을 지정하여 출력 가능.
-> 시스템에 따라 적용되지 않을 수 있음.

형변환자	뒤에 올 수 있는 변환 문자	변환 형
hh	d, i, o, u, x, X	signed/unsigned char
h	d, i, o, u, x, X	signed/unsigned short
l	d, i, o, u, x, X	signed/unsigned long
l	c	wint_t
l	s	wchar_t
l	a, A, e, E, f, F, g, G	무시
ll	d, i, o, u, x, X	signed/unsigned long long
j	d, i, o, u, x, X	intmax_t나 uintmax_t
z	d, i, o, u, x, X	size_t
t	d, i, o, u, x, X	ptrdiff_t
L	a, A, e, E, f, F, g, G	long double

메모 포함[이94]: ex.
printf("i = %8d₩n", i);
printf("i = %*d₩n", 8, i);
이 두 문장은 동일한 기능을 한다.

인자로 작성한 8이 *에 들어간다.
기본적으로 인자에 작성한 값은 변환명세의 변환문자로 들어간다고 생각하는 것이 간단할 듯.

메모 포함[이95]: 사용 방식은 field-width의 것과 동일하다.

3. scanf()

표준 입력 장치로 데이터를 입력 받는 함수.
첫 번째 인자에 명시한 제어 문자열 대로 데이터를 입력 받음.
enter 키가 입력될 때까지 입력받음. (%s 의 경우 공백문자 또는 enter 키.)
성공적으로 데이터를 입력 받은 변환 명세의 개수를 리턴하고, 파일의 끝을 만나면 EOF 를 리턴함.

메모 포함[이96]: 읽을 것이 없으면 0을 리턴한다.
읽을 것이 없는 것과 파일의 끝을 만난 것은 다르다.

오류로 아무 값도 입력 받지 못한 경우 0을 리턴한다.

1) 인자 작성법

첫번째 인자에는 입력받을 형태인 제어문자열이 들어감.
제어 문자열은 변환 명세와 일반 문자들로 구성됨.

두번째 인자부터는 데이터를 저장할 변수의 주소나 메모리 공간의 주소가 들어감.
하나의 함수에서 여러 개의 값을 내보내기 위해 주소를 작성하는 것.
인자의 개수에는 제한이 없음.

```
int scanf(const char * restrict format, . . . );
```

2) 작동 방식

제어 문자열을 읽다가 변환 명세를 찾으면 해당 형식대로 데이터를 읽어 해당 변수에 저장함.
제어 문자열을 읽다가 일반 문자를 찾으면 입력 스트림에서 동일한 문자를 제거함.
제어 문자열의 일반 문자가 입력 스트림과 동일하지 않으면 입력 오류로 처리하여 리턴값을 보내고 종료함.
즉, scanf()의 제어 문자열에 일반 문자를 작성했다면, 데이터 입력 시에도 동일한 형식으로 작성해야 함.

메모 포함[이97]: ex.
scanf("name : %s", s); 인 경우, 입력 스트림의 값이 name : 으로 시작하지 않는다면 %s로 데이터를 입력 받을 수 없다.

2) 변환 명세

값을 어떤 형식으로 출력할 것인지 명시하는 키워드.
%로 시작하여 변환문자로 끝남.
출력할 데이터와 변환명세를 반드시 맞춰 줘야 함. 자동변환 없음.

메모 포함[이98]: 경고 메시지가 나오고 이상한 값이 출력됨.
자료형이 달라도 큰 문제가 없는 경우에는 정상적인 값이 나오기도 함.

c	공백을 포함한 모든 문자	char 포인터
d, i	10진 정수 (부호는 옵션)	정수 포인터
u	10진 정수 (부호는 옵션)	부호없는 정수 포인터
o	8진 정수 (부호는 옵션)	부호없는 정수 포인터
x	16진 정수 (부호는 옵션)	부호없는 정수 포인터
a, e, f, g	실수 (부호는 옵션)	실수 포인터
s	공백 없는 문자열	char 포인터
p	보통 16진 정수 (시스템에 따라 다름)	void 포인터
n	지금까지 읽은 문자 개수를 대응 인자에 배정, 입력 스트림의 내용은 안 읽음	정수 포인터
%	입력 스트림에서 % 읽음	대응 인자 없음
[ ]	다음에 설명	char 포인터

%c 는 field-width 만큼의 문자를 읽어서 해당 주소에 저장함. 이 경우 해당 주소는 충분한 공간을 가져야 함.
%s 는 입력 스트림에서 공백문자/개행문자까지의 문자열을 읽어서 NULL 문자를 끝에 추가하여 해당 주소에 저장함. 이 경우 해당 주소는 충분한 공간을 가져야 함.

%[]는 입력 스트림에서 스캔 집합 이외의 문자가 나올 때까지 문자열을 읽어서 해당 주소에 저장함.
스캔 집합의 문자들만 입력받은 후, 맨 끝에 NULL 문자를 추가함.
스캔 집합 : 원하는 문자로만 구성된 집합.
[] 내의 첫 번째 문자가 ^인 경우에는 괄호 내 문자를 제외한 문자들이 스캔 집합이 됨.
[] 내의 첫 번째 문자가 ^가 아닌 경우에는 괄호 내의 문자들이 스캔 집합이 됨.
대쉬(-)를 사용해서 명시할 수도 있음.

메모 포함[이99]: ex.
scanf("%[^₩n₩t]", s); -> 공백, 개행, 탭을 제외한 문자들만 읽는다. 공백, 개행, 탭이 나오면 읽기를 중단한다.

메모 포함[이100]: ex.
scanf("%[0-9a-fA-F]", s); -> 0~9, a~f, A~F의 문자열만 읽는다. 이외의 문자가 나오면 읽기를 중단한다.

3) 변환 명세 옵션

%와 변환문자 사이에 *, field-width, length-modifier를 순서대로 작성해 옵션을 부여함.

* -> 입력 스트림의 값을 가져와서 버리는 옵션.

field-width -> 읽어드릴 필드의 최대 크기를 지정하는 옵션.
  -> 스트림에서 한 번에 가져올 크기를 지정하는 것.
  -> 0 이 아닌 정수로 명시해야 함.

length-modifier -> 형변환자. 인자 출력 시 자료형을 지정하여 출력 가능.
  -> 시스템에 따라 적용되지 않을 수 있음.

형변환자	뒤에 올 수 있는 변환 문자	인자의 형
hh	d, i, o, u, x, X, n	signed/unsigned char 포인터
h	d, i, o, u, x, X, n	signed/unsigned short 포인터
l	d, i, o, u, x, X, n	signed/unsigned long 포인터
l	c, s, [ ]	wchar_t 포인터
l	a, A, e, E, f, F, g, G	double 포인터
ll	d, i, o, u, x, X, n	signed/unsigned long long 포인터
j	d, i, o, u, x, X, n	intmax_t나 uintmax_t 포인터
z	d, i, o, u, x, X, n	size_t 포인터
t	d, i, o, u, x, X, n	ptrdiff_t 포인터
L	a, A, e, E, f, F, g, G	long double 포인터

메모 포함[이101]: ex.
scanf("%d %*d, %d", &a, &b); 인 경우,
입력 스트림이 20 30 40이면
a와 b에는 각각 20, 40이 들어간다.
30은 버려진다.

메모 포함[이102]: ex.
scanf("%3d %5c", &a, &c); 인 경우,
입력 스트림이 1234567890이면,
a에는 123이 정수로 저장되고, 배열 c에는 45678이 각각 문자로 저장된다.

4) 주의점

오류로 인해 scanf()가 중간에 중단된 경우, 값들이 버퍼에 살아 있어 이후의 코드에 영향을 미칠 수 있음.
scanf()의 리턴값을 검사하는 코드를 작성하는 것이 좋음.

메모 포함[이103]: ex.
```
if(scanf("%d%d", &a, &b) != 2)
{
  printf("오류 발생!");
}
```

4. sprintf() / sscanf()

표준 입출력 장치 대신 문자열에 입/출력하는 함수들.

첫 번째 인자에 입/출력할 문자열을 지정하는 것을 제외하면, printf(), scanf()와 동일한 방식으로 사용함.

1) 사용 방식

문자열 처리 시 사용함.

1. 문자열을 다른 형으로 바꾸기.
sscanf()로 문자열을 입력 받아서 다른 변수에 저장하여 다른 형으로 바꿀 수 있음.
정수 자료형 등에 저장하여 atoi() 함수를 대체할 수 있음.

2. 문자열을 여러 개의 부분으로 나누기.
sscanf()로 문자열을 입력 받아서 여러 변수에 저장하여 여러 부분으로 나눌 수 있음.

```
int printf(char * restrict s, const char * restrict format, . . . );
int scanf(char * restrict s, const char * restrict format, . . . );
```

2) 주의점

scanf()와는 달리, sscanf()는 버퍼를 사용하지 않기 때문에 sscanf()는 호출될 때마다 문자열의 처음부터 읽음.
즉, 오류가 발생해도 sscanf()끼리 영향을 주고받지 않음.

sprintf() 또한 호출될 때마다 문자열의 처음부터 입력함.

7. 파일

1. 파일 입출력

1. 파일 포인터

프로그램에서는 파일 이름 대신 해당 파일을 가리키는 파일 포인터를 사용함.

FILE * 형으로 정의함.

FILE 형은 stdio.h 에 들어 있는 구조체로, 파일의 현재 상태를 나타내는 멤버들을 가지고 있음.

stdio.h 에 정의된 stdin, stdout, stderr 도 각각 표준 입출력 장치를 가리키는 파일 포인터임.

2. fopen()

인자로 지정한 파일을 지정한 모드로 열어서 파일 포인터 값을 리턴하는 함수.

파일로 데이터를 입출력할 때에는 우선 해당 파일을 열어야 함.

파일을 연 이후부터는 파일을 문자 스트림처럼 사용할 수 있음.

메모 포함[이104]: 파일을 열기 전에는 파일 내용을 출력하거나 파일에 입력할 수 없다.

파일을 여는 것은 덮인 책을 여는 것과 같다.

1) 인자 작성법

첫 번째 인자에는 파일 이름을 지정함.

해당 파일은 실행 파일과 동일한 디렉토리에 위치해야 함.

```
FILE *fopen(const char * restrict filename,
            const * char restrict mode);
```

메모 포함[이105]: 그렇지 않은 경우도 있는지는 모르겠다.

두 번째 인자에는 모드(파일 열기 방식)을 지정함.

파일 이름과 모드는 각각 문자열로 지정함.

2) 작동 방식

해당 파일 이름을 가진 파일을 해당 파일 열기 방식으로 연 후, 해당 파일의 파일 포인터 값을 리턴함.

파일을 여는 데에 실패하면 NULL 을 리턴함.

3) 모드 종류

"r"	읽기를 위해 텍스트 파일 열기
"w"	쓰기를 위해 텍스트 파일의 내용을 삭제 후 열기, 없는 파일이면 생성 후 열기
"wx"	배타적 쓰기를 위해 텍스트 파일을 생성 후 열기
"a"	첨가; 파일의 끝에 쓰기를 위해 텍스트 파일 열기, 없는 파일이면 생성 후 열기
"rb"	읽기를 위해 이진 파일 열기
"wb"	쓰기를 위해 이진 파일의 내용을 삭제 후 열기, 없는 파일이면 생성 후 열기
"wbx"	배타적 쓰기를 위해 이진 파일을 생성 후 열기
"ab"	첨가; 파일의 끝에 쓰기를 위해 이진 파일 열기, 없는 파일이면 생성 후 열기
"r+"	업데이트(읽기와 쓰기)를 위해 텍스트 파일 열기
"w+"	업데이트(읽기와 쓰기)를 위해 텍스트 파일의 내용을 삭제 후 열기, 없는 파일이면 생성 후 열기
"w+x"	배타적 업데이트(읽기와 쓰기)를 위해 텍스트 파일 열기
"a+"	첨가; 파일의 끝에 쓰기를 위해 업데이트를 위한 텍스트 파일 열기, 없는 파일이면 생성 후 열기
"r+b" / "rb+"	업데이트(읽기와 쓰기)를 위해 이진 파일 열기
"w+b" / "wb+"	업데이트(읽기와 쓰기)를 위해 이진 파일의 내용을 삭제 후 열기, 없는 파일이면 생성 후 열기
"w+bx" / "wb+x"	배타적 업데이트(읽기와 쓰기)를 위해 이진 파일을 생성 후 열기
"a+b" / "ab+"	첨가; 파일의 끝에 쓰기를 위해 업데이트를 위한 이진 파일 열기, 없는 파일이면 생성 후 열기

r 을 지정한 경우 파일이 존재하지 않으면 파일 열기에 실패함.

b 가 붙은 것들은 이진 파일을 위한 것들.

a 를 지정한 경우 해당 파일의 맨 끝부터 작성을 시작함.

+가 붙은 것들은 읽기와 쓰기를 위한 것들.
하나의 파일에 대해 읽기와 쓰기를 동시에 하는 경우에는, 읽기 작업과 쓰기 작업 사이에 fseek(), rewind() 등을 사용한 데이터 처리를 반드시 해줘야 함.
해당 파일을 닫고 다시 여는 방법으로도 처리가 가능함.

x 가 붙은 것들은 배타적 파일 접근을 위한 것들.
x 를 사용한 경우 파일이 이미 존재하거나 파일 생성이 불가능하면 오류가 발생함.
해당 프로그램만이 해당 파일에 접근할 수 있게 됨.

이것들 이외의 모드를 지정하면, 시스템에 따라 반영 방식이 다를 수 있음.

메모 포함[이106]: 파일 위치 지시자를 맨 처음으로 이동시켜야 정상적으로 작동한다.

4) 주의점

파일이 열리지 않은 경우에 대해 예외 처리를 해 주는 것이 좋음.

메모 포함[이107]: ex.

```
if(file != fopen("file.txt", "r"))
{
  printf("error₩n");
}
```


3. fclose()

인자로 지정한 파일 포인터에 해당하는 파일을 닫는 함수.

파일의 사용이 끝나면 해당 파일을 닫아야 함.

```
int fclose(FILE *stream);
```

1) 인자 작성법

인자에 닫을 파일의 파일 포인터를 지정함.

4. 파일 포인터 함수들

```
int getc(FILE *stream);
int putc(int c, FILE *stream);
```

1) getc() / putc()

파일에 문자를 입출력하는 함수들.

마지막 인자에 파일 포인터를 지정하는 것을 제외하면 getchar() / putchar() 와 동일하게 사용함.
getchar()와 getc(stdin), putchar('c')와 putc('c', stdout)은 동일한 기능을 함.

2) fprintf() / fscanf()

파일에 문자열을 출력하고, 파일의 데이터를 입력받는 함수들.

첫 번째 인자에 파일 포인터를 지정하는 것을 제외하면 printf() / scanf() 와 동일하게 사용함.
printf(. . .)와 fprintf(stdout, . . .), scanf(. . .)와 scanf(stdin, . . .)은 동일한 기능을 함.

```
int printf(FILE * restrict stream, const char * restrict format, . . . );
int scanf(FILE * restrict stream, const char * restrict format, . . . );
```

+ getc()로 파일 전체 내용 읽기

getc()로 파일의 전체 내용을 아주 간단히 읽을 수 있음.

```
while((c = getc(file)) != EOF)
{
        <명령>
}
```

이 반복문 하나면 파일의 전체 내용을 꺼낼 수 있음.
EOF 를 유용하게 사용하자.

+ stderr 사용

오류 메시지를 코드로 출력할 때에는 주로 stderr 를 파일 포인터로 지정함.

프로그램 내에서 오류 메시지를 stdout 으로 지정한 경우 리다이렉션 등의 기능을 사용할 때 의도와는 달라질 수 있으므로 유의해야 함.

2. 파일 위치 지시자

1. 파일 위치 지시자

현재 파일을 읽거나 쓸 지점을 가리키는 지시자.
파일을 읽거나 쓰는 과정에서 파일 위치 지시자는 해당 부분으로 이동함.

파일 위치 지시자는 파일 입출력 함수가 종료된 후에도 유지되기 때문에, 파일 입출력 함수들은 마지막으로 입출력이 일어난 지점부터 다음 입출력을 진행하게 됨.

파일 위치 지시자의 위치를 지정하여 파일의 임의의 위치에 접근할 수 있음.

메모 포함[이108]: 텍스트 파일과 이진 파일 모두에서 사용이 가능하다.

2. 파일 위치 지시자 관련 함수들

1) ftell(<파일포인터>)

파일 위치 지시자의 위치 값을 리턴하는 함수.
파일의 처음부터 몇 바이트 떨어진 곳에 파일 위치 지시자가 위치하는지를 리턴함.

인자로 확인할 파일의 파일 포인터를 지정함.

2) fseek(<파일포인터>, <offset>, <place>)

파일 위치 지시자의 값을 직접 지정하는 함수.

파일의 내용을 다루는 위치를 place 값으로부터 offset 바이트 만큼 떨어진 곳으로 지정함.
offset 으로 양수를 지정하면 파일의 뒤쪽으로 이동하고, 음수를 지정하면 파일의 앞쪽으로 이동함.
0 을 지정하면 이동하지 않음.

place 값.

SEEK_SET	-> 0 을 나타내는 기호 문자.
	-> 파일의 시작을 의미함.
SEEK_CUR	-> 1 을 나타내는 기호 문자.
	-> 현재 위치를 의미함.
SEEK_END	-> 2 를 나타내는 기호 문자.
	-> 파일의 끝을 의미함.

3) rewind(<파일포인터>)

파일 위치 지시자를 파일의 시작으로 설정하는 함수.

rewind(FILE_ptr)은 fseek(FILE_ptr, 0, SEEK_SET)과 동일한 기능을 함.

+ 이진 파일의 크기 구하기

fseek() 함수로 파일의 끝으로 이동한 다음 ftell() 함수로 구한 위치가 해당 이진 파일의 크기임.
여기서 파일의 크기는 기본적으로 바이트 단위로 구할 수 있음.

3. 텍스트 파일과 이진 파일

1. 텍스트 파일

텍스트로 쓰인 파일.

putc()와 fprintf()는 파일에 문자나 문자열로 출력함.

1) 장점

문서 편집기로 내용을 쉽게 파악할 수 있음.

2) 단점

프로그램에서 데이터에 접근하기 까다로움. -> 이진 파일로 저장하여 해결할 수 있음.

메모 포함[이109]: 같은 메모리 크기를 가지는 정보들이라도, 문자로 출력하면 그 크기가 제각각이기 때문에 해당 정보들은 꺼내기가 상당히 번거로워진다.

2. 이진 파일

메모리 내용과 같은 형식으로 작성된 파일.

메모리 내용과 같은 형식으로 작성되었기 때문에 동일한 자료형의 정보들은 동일한 길이를 가짐.
즉, 데이터를 편리하게 관리할 수 있음.

프로그램 내에서 이진 파일은 fwrite() 함수와 fread() 함수를 사용해 다룸.

+ 이진 파일의 내용을 편집기로 확인하기

이진 파일을 확인하면, 알파벳은 아스키 값으로 저장되기 때문에 시각적으로 확인할 수 있음.
정수는 2 의 보수로 저장되기 때문에 시각적으로 확인할 수 없음.
알파벳의 아스키 값과 동일한 정수는 알파벳으로 확인할 수 있음.

3. fwrite()

메모 포함[이110]: ex.
fwrite(file_ptr, sizeof(char), 1, myfs);

데이터를 메모리 형태로 파일에 입력하는 함수.

1) 인자 작성법

ptr -> 파일에 출력할 데이터를 가지고 있는 포인터.
size -> 각 원소의 크기.
nmemb -> 원소의 개수.
stream -> 데이터를 저장할 파일 포인터.

```
size_t fwrite(const void * restrict ptr, size_t size,
    size_t nmemb, FILE * restrict stream);
```

2) 작동 방식

ptr 이 포인트하는 곳에서 시작해서, size 만큼의 크기를 가지는 nmemb 개의 원소를 stream 에 메모리 형태로 입력함.

4. fread()

메모리 형태로 쓰여진 이진 파일에서 데이터를 입력받는 함수.

1) 인자 작성법

ptr -> 파일에서 입력받은 데이터를 저장할 메모리 공간의 포인터.
size -> 각 원소의 크기.
nmemb -> 원소의 개수.
stream -> 데이터를 읽어올 파일 포인터.

```
size_t fread(const void * restrict ptr, size_t size,
    size_t nmemb, FILE * restrict stream);
```

2) 작동 방식

size 만큼의 크기를 가지는 nmemb 개의 원소를 stream 에서 입력받아 ptr 이 포인트하는 곳에 저장함.

3) 주의점

이진 파일을 읽어서 저장할 때는, 저장할 메모리 공간이 저장할 데이터와 동일한 자료형을 가지고 있어야 오류가 발생하지 않음.
이진 파일에 데이터를 입력할 때 사용했던 자료형을 그대로 사용하는 것이 좋음.

기타

1. 변수 선언 옵션들

1. 형 한정자

변수의 사용 제한 설정

1) const

const 한정자가 사용되면 선언 시에 초기화하는 것을 제외하고는 값을 수정할 수 없다.

1. 일반 변수에서의 사용 -> 자료형의 앞에 작성함. (ex. const char a[] = "Hello~!";)
-> 배열에도 지정 가능.

2. const int * ptr; -> 1. const 일반 변수 포인트 가능 2. 값 수정 가능 3. 역참조 수정 불가능
-> 일반 포인터 변수는 const 를 사용한 일반 변수를 포인트할 수 없음.
포인터 변수의 자료형에도 const 를 넣어야 해당 변수를 포인트할 수 있음.
-> 이 const 포인터 변수는 그것 자체의 값을 수정할 수 있음.
-> 이 const 포인터 변수를 역참조하여 접근한 메모리의 값은 수정할 수 없음.

2. int * const ptr; -> 1. const 일반 변수 포인트 불가능 2. 값 수정 불가능 3. 역참조 수정 가능
-> 포인터의 값을 수정할 수 없게 하려면, 선언 시에 *와 식별자 사이에 const 를 작성함.
-> 이 const 포인터를 역참조하여 접근한 메모리의 값은 수정할 수 있음.

3. const int * const ptr; -> 1. const 일반 변수 포인트 가능 2. 값 수정 불가능 3. 역참조 수정 불가능
-> 앞, 뒤에 모두 const 를 작성하면 2, 3 번의 기능이 모두 적용됨.

정리하면,
일반 변수에서는 자료형 앞에 작성함.
그 변수를 가리키려면 포인터에도 자료형 앞에 작성함. -> 포인터는 변경 가능, 역참조는 변경 불가능.
포인터 자체를 막으려면 자료형 뒤에 작성함. -> 포인터 변경 불가능, 역참조는 변경 가능.

메모 포함[이111]: ex.
const int a = 10; 인 경우, const int *ptr = & a;
사용이 필요하다.

2) restrict

코드 최적화를 위한 형 연산자. (실행 시간 단축)

포인터 변수에 사용함.
선언 시에 * 뒤에 작성함.

컴파일러에게 그 포인터만을 통해 현재 포인트하는 객체가 접근됨을 알려줌.
(해당 객체를 최대 하나의 포인터로만 포인트할 것이라는 의미.)
restrict 가 붙은 포인터끼리는 포인트하는 대상이 겹치지 않을 것으로 가정하고 최적화함. 겹치면 오류 발생.

restrict 가 붙은 포인터들을 사용하여 서로 배정해도 오류 발생.

메모 포함[이112]: 배정을 선언할 때 해보는 걸로 실험해보자.

3) volatile

객체가 프로그램 내의 코드가 아니라 외부 사건에 의해 변경될 수 있음을 컴파일러에게 알리기 위해 사용하는 형 연산자.

외부 사건의 영향을 받는다면 이거 써줘야 함. 이유는 최적화 방식과 관련이 있음.

일반 변수에서는 자료형의 앞에 작성하든 뒤에 작성하든 똑같음.

포인터에서는 위치에 따라 달라짐.
1. volatile int *ptr; -> ptr 이 포인트하는 곳에 volatile 이 적용됨.
2. int * volatile ptr; -> ptr 에 volatile 이 적용됨.
3. volatile int * volatile ptr; -> ptr 이 포인트하는 곳과 ptr 에 volatile 이 적용됨.

+ 포인터에서 형 지정자 사용 위치

포인터 자체에 적용하려면 * 뒤에 작성함.
포인터가 가리키는 곳에 적용하려면 맨 앞에 작성함.

3. 정렬 지정자

1) 정렬

데이터가 메모리의 특정 주소에 위치하는 것.

c 에서는 변수 선언 시에 정렬 방식을 지정할 수 있음.

2) 정렬 요구조건 속성

객체가 몇 바이트 단위로 메모리에 할당되는지를 나타내는 정수 값.

대체로 해당 객체의 크기와 정렬 요구조건 속성은 동일함.

정렬 요구조건 속성이 n 이면, n 의 배수의 주소 값을 가지는 메모리에 데이터가 정렬됨.

3) _Alignof() 연산자

인자에 지정한 객체의 정렬 요구조건 속성 값을 알려주는 연산자.

```
_Alignof(<객체>)
```

_Alignof() 연산자의 괄호 안에는 자료형, 수식이 올 수 있음.

자료형이 사용될 때는 괄호가 반드시 필요함.
자료형이 아닌 객체들은 괄호 생략 가능.
(sizeof()와 사용 형식 유사.)

4) 정렬 지정자

선언문에서 정렬 속성을 지정하는 지정자.
정렬 속성은 2 의 거듭제곱수여야 함.

함수, 매개변수, register 에는 사용 불가.

1. _Alignas(자료형) <자료형> <변수명>
해당 자료형의 정렬 요구조건 속성을 현재 선언하는 변수의 정렬 요구조건 속성으로 함.

2. _Alignas(<2 의 거듭 제곱수(0 제외)>) <자료형> <변수명>
해당 정수를 현재 선언하는 변수의 정렬 요구조건 속성으로 함.

+ 더 간단한 표기

정렬을 위해 <stdalign.h> 헤더파일 제공.
alignas() -> _Alignas()와 동일.
alignof() -> _Alignof()와 동일.

2. 가변 인자

1. 가변 인자 함수

매개변수의 개수가 정해져 있지 않은 함수.

1) 정의 / 원형

매개변수 목록의 맨 마지막 인자 뒤에 . . . 을 작성함.

<stdarg.h> 헤더파일에 정의된 매크로를 사용하여 가변 인자 함수를 제작함.
(standard argument. 표준 인자.)

2) 가변 인자 함수 제작 매크로들

<stdarg.h> 헤더파일에 정의된 매크로들.

ap 에 지정할 변수는 va_list 형으로 작성함.
va_list 형으로 작성한 변수는 va_start() 매크로로 초기화해야 함.

1. va_start(ap, v)
-> ap 가 가변 인자의 첫 번째 인자를 포인트하게 하는 매크로.
-> v 는 . . . 바로 직전 매개변수의 식별자임.

2. va_arg(ap, type)
-> ap 가 포인트하는 곳의 값을 type 형으로 읽고, ap 가 다음 인자를 포인트하게 하는 매크로.

3. va_copy(dest, src)
-> 가변 인자를 포인트하는 src 포인터를 dest 로 복사하는 매크로.

4. va_end(ap)
-> va_start() 매크로로 초기화된 ap 포인터를 제거하는 매크로.
-> va_start() 매크로를 사용한 함수에서 사용해야 함.

가변 인자 함수에서는 가변 인자 이외의 형을 가진 인자가 1 개 이상이어야 함.
가변 인자 함수에서는 . . . 에 위치할 인자의 자료형을 알아야 함.
가변 인자 함수에서는 몇 번째 인자가 마지막인지를 알 수 있어야 함.

3) 호출

인자의 개수가 가변적인 것을 제외하면, 일반적인 함수와 동일하게 호출함.
매개변수 목록에서 . . . 이 위치한 곳에는 인자를 작성하지 않거나 1 개 이상 작성하여 함수를 호출함.

메모 포함[이113]: ex.
int printf(const char *cntrl_string, . . .);

메모 포함[이114]: 이거는 제대로 설명 안 해줬다.

메모 포함[이115]: printf() 함수에서는 제어 문자열에 있는 변환 명세로 형을 알아낸다.

인자에 형과 값을 모두 쓰도록 하는 방법도 있다.

메모 포함[이116]: 마지막 인자에는 0이나 기호 상수 등을 입력하여 끝임을 알리는 방법도 있다.

2. 가변 인자 매크로

가변적인 개수의 매개변수들을 갖는 매크로.

매개변수 목록에는 . . . 을 작성함.
매개변수 목록에 작성한 인자들이 들어갈 위치는 __VA_ARGS__로 명시함.

작성한 인자들은 __VA_ARGS__의 위치에 대체되어 들어감.

가변 인자 매크로는 에러 메시지를 출력할 때 종종 사용됨. (파일명, 행, 함수 등 반영 표시 가능.)

메모 포함[이117]: ex.
#define TEST(. . .) printf("%d %d₩n", __VA_ARGS__);
이 경우,
TEST(a, b)는 printf("%d %d₩n", a, b); 로 대체된다.

+ 다중 줄 매크로

여러 줄로 이루어진 매크로는 콤마 연산자(,)와 ₩을 사용하여 작성함. (하나의 수식으로 만듦.)
각 줄의 끝에 ,₩을 작성함.
, 대신 ; 을 사용하면 세 개의 서로 다른 수식으로 취급되어 문제가 발생할 수 있음.

또는 do-while 문을 사용해 여러 문장들을 하나로 묶기도 함.
do-while 문의 중괄호 안에 문장들을 넣고, while 문의 조건에 0 을 넣어 한 번만 실행되도록 하는 것.

이렇게 하나의 수식으로 묶거나 괄호로 묶어서 사용하는 것은 연산자 수식을 괄호로 묶는 것과 유사한 맥락임.

메모 포함[이118]: ex.
#define TEST(a, b) printf("%d₩n", a),₩
printf("%d", b);

3. 진단 코드

1. 진단 코드

특정한 조건에 따라서 프로그램의 실행 여부를 결정할 때 사용하는 기능.
조건에 따라서 실행 중인 프로그램을 종료하기도 함.

디버깅 시에 유용하게 사용됨.

#error, assert(), _Static_assert() 중 하나로 진단 코드를 사용함.

2. #error

프로그램 컴파일 시에 전처리기가 #error 전처리 지시자를 만나면 오류 메시지를 출력하고 컴파일을 중단함.
해당 프로그램의 실행 파일이 생성되지 않음.

#error 뒤에 작성한 문자열을 포함한 오류 메시지가 출력됨.
이 오류 메시지의 형태는 컴파일러마다 다름.

```
#error <문자열>
```

3. assert()

조건을 검사하여 해당 프로그램이 조건에 맞지 않으면 진단 메시지를 출력한 후 abort()를 출력하여 프로그램을 종료하는 함수.

```
assert(<조건 수식>)
```

<assert.h> 헤더파일을 include 해야 사용할 수 있음.
<assert.h>가 include 되기 이전에 NDEBUG 매크로가 정의되어 있으면 assert()는 무시됨.
(#define NDEBUG)

조건 수식이 참이면 아무 작업도 하지 않음.
조건 수식이 거짓이면 진단 메시지를 출력하고 abort() 함수를 출력하여 프로그램을 종료함.
진단 메시지로는 주로 파일명, 함수 식별자, 행 번호 등이 출력됨.

4. _Static_assert()

컴파일 시에 조건을 검사하여 조건에 맞지 않으면 오류 메시지를 출력하고 컴파일을 중단시키는 함수.
해당 프로그램의 실행 파일이 생성되지 않음.

#error 는 전처리 과정에서 전처리기가 처리하지만, _Static_assert()는 전처리 과정이 끝난 다음의 컴파일 과정에서 처리됨.
#error 는 전처리기가 처리하기 때문에 c 연산자를 사용할 수 없음.
_Static_assert()는 전처리 이후에 처리하기 때문에 c 연산자를 사용할 수 있음.

컴파일 시 첫 번째 인자로 작성한 조건 수식이 참이면 아무 작업도 수행하지 않음.
컴파일 시 첫 번째 인자로 작성한 조건 수식이 거짓이면 두 번째 인자에 작성한 문자열을 포함한 진단 메시지를 출력함.

<assert.h>를 include 하면 _Static_assert() 대신 static_assert()로 사용할 수 있음.

```
_Static_assert(<정수 상수 수식>, <문자열>)
```

+ abort() 함수와 core dumped

abort() 함수로 프로그램을 종료하면 Abort (core dumped)라는 메시지가 출력됨.
이것은 프로그램이 예기치 못하게 종료되었고, 해당 프로그램의 메모리 내용을 파일로 저장했음을 나타냄.
해당 파일은 디버깅 프로그램으로 확인할 수 있음.

+ assert() 함수 심화

assert() 함수의 인자에 작성하는 조건 수식에 "&& 문자열"을 붙여서 프로그램 강제 종료 시 해당 문자열로 강제 종료 사유를 더 편리하게 확인할 수 있음.
전체 조건 수식의 참/거짓은 원래의 조건 수식에 의해 결정됨.

4. 신호

1. 프로세스

실행 중인 프로그램.

2. 신호

프로세스에게 정보를 저장하는 가장 단순한 방법.
외부 환경의 변화나 오류 등을 전달함.

일반적으로 신호는 비정상적인 상황에서 생성되고, 프로세스를 종료시킴.

사용자가 키를 입력하여 신호를 전달할 수도 있음.
하나의 프로세스가 다른 프로세스에 신호를 전달하기도 함.

신호 관련 매크로와 함수들은 <signal.h> 헤더파일에 들어 있음.

메모 포함[이119]: ex.
0으로 나눈 경우.
잘못된 메모리 참조.

메모 포함[이120]: ex.
ctrl + c로 프로세스 종료.

3. 신호 관련 기본 매크로

운영체제가 다루는 신호들에 대한 매크로들.
신호 관련 매크로는 SIG 로 시작함.

신호는 내부적으로 정수 상수임.

```
#define SIGINT    2     : 인터럽트. (ctrl + c 입력 시 발생하는 신호.)
#define SIGILL    4     : 비정상 연산.
#define SIGFPE    8     : 부동 소수점 예외.
#define SIGKILL   9     : KILL.
#define SIGSEGV   11    : 세그먼트 위반.
#define SIGALRM   14    : 알람 클락.
```

이 신호들은 대부분의 시스템이 다룰 수 있는 것들임.

4. 신호 처리 매크로

이 매크로들은 signal() 함수의 두 번째 인자로 지정함.

SIGKILL(9) 신호는 사용자가 임의로 다룰 수 없음. -> SIGKILL(9) 신호를 받은 프로세스는 무조건 종료됨.

1) SIG_DFL

신호가 발생했을 때 그 신호에 해당하는 default 작업을 수행하게 함.

```
#define SIG_DFL ((void (*) (int)) 0)
```

메모 포함[이121]: signal default

2) SIG_IGN

신호가 발생했을 때 해당 신호를 무시하게 함.

```
#define SIG_DFL ((void (*) (int)) 1)
```

메모 포함[이122]: signal ignore

5. 신호 관련 함수

SIGKILL(9) 신호는 사용자가 임의로 다룰 수 없음. -> SIGKILL(9) 신호를 받은 프로세스는 무조건 종료됨.

1) signal()

```
void (*signal(int sig, void (*func)(int)))(int);
```

신호를 수신한 경우 지정한 함수를 실행하는 함수.

이 함수의 함수 원형은 함수 포인터를 리턴하기 때문에 이런 형식을 가짐.

첫 번째 인자에는 신호의 종류를 지정함. (주로 매크로로 지정함.)
두 번째 인자에는 실행할 함수의 식별자를 지정함.
두 번째 인자에 작성한 함수의 인자에는 첫 번째 인자에 작성한 신호의 종류가 지정됨.
두 번째 인자에는 함수 대신 SIG_DFL, SIG_IGN 매크로를 사용할 수도 있음. 이 경우 해당되는 작업이 수행됨.

기본적으로 신호를 수신하면 해당 프로세스는 종료되는데, 이 함수를 사용한 경우 종료하는 것 대신에 해당 함수를 실행함.

해당 함수에는 신호가 입력된 경우 수행할 작업들을 작성함.

+ SIGABRT

abort() 함수는 SIGABRT 신호를 보내 프로그램을 종료시킴.

5. 프로그래밍 도구

1. gcc 컴파일러

gcc <option> <file-list>

c 프로그램을 위한 컴파일러.

1) option

-save-temps : 컴파일 단계 별 결과물을 모두 파일로 저장.

옵션	설명
--help	gcc 옵션에 대한 간략한 설명 출력
-c	링크를 하지 않고 컴파일만 수행, 컴파일 결과는 .o 파일로 생성됨
-D id[=value]	id 매크로를 정의함, value가 없으면 id는 1의 값을 가짐
-E	전처리기만 수행함, 전처리 결과는 화면에 출력됨
-g	gdb 디버거가 사용할 수 있는 디버깅 정보를 만드는 코드 생성
-I dir	dir 디렉터리에서 헤더파일을 찾음
-L dir	라이브러리 패스에 dir 디렉터리를 추가함
-llib	lib로 명시된 라이브러리를 찾음
-M	Makefile을 만들 때 유용한 종속 항목 출력
-o name	실행 파일 이름을 name으로 함
-Olevel	코드 최적화 수준을 선택함, level: 0, 1, 2, 3, s
-pg	gprof 프로파일러가 사용할 수 있는 프로파일 정보를 만드는 코드 생성
-S	어셈블러 코드 생성, .s 파일로 생성됨
-std=standard	어떤 표준으로 작성된 프로그램인지 지정함, standard: c18, c11, c99, c89 ansi 등

2) file-list

하나 이상의 파일명을 명시함.

.c .h .o 확장자를 가지는 파일들만 작성할 수 있음.

.c : c 소스 파일

.h : 헤더파일

.o : 목적 파일

메모 포함[이123]: gcc 컴파일러는 우분투에서 사용하는 것이지만, 컴파일러 사용 원리는 대체로 비슷해서 여기서 정리한 내용을 자신이 사용하는 컴파일러에 적용해 볼 수 있다.

3) gcc 컴파일러 작동 과정

4 가지 과정을 거침.

1. 전처리기(-E) : 소스 코드의 전처리 수행.
2. 컴파일러(-S) : 전처리가 수행된 소스 코드를 어셈블리 코드로 변환.
3. 어셈블러(-c) : 어셈블리 코드를 목적 코드로 변환
4. 링커 : 목적 코드로 실행 프로그램 생성.

gcc 의 옵션들을 사용하여 각 단계별로 작업을 직접 수행할 수 있음.

메모 포함[이124]: 아직 실행할 수 없다. 목적 코드(파일)을 사용하는 것은 작업 과정의 효율을 높이기 위함이다.

목적 파일을 개별적으로 만들면 필요한 부분만 따로 컴파일할 수 있다.

2. make

여러 개의 파일로 작성된 프로그램을 효율적으로 컴파일하기 위한 도구.

1) make 파일

make 를 사용하기 위해서는 make 파일을 만들어야 함.

```
<target> : <dependencies>
        <commands>
```

해당 파일의 이름은 makefile, Makefile, GNUmakefile 등으로 지정할 수 있음

makefile 은 규칙들로 이루어진 나름의 형식을 가짐.

- target -> 라벨 역할을 하는 것으로, 해당 규칙에 의해 생성되는 파일의 이름을 작성함.
- dependencies -> target 을 만들기 위해 필요한 파일들을 작성함.
- commands -> target 을 만들기 위해 실행되는 명령
  - -> tab 으로 시작해야 함.
  - -> tab 을 여러 번 사용하여 하나의 규칙에 여러 개의 명령을 작성할 수 있음.

2) make 실행

make 또는 make <target 이름>을 입력하여 make 를 실행함.
make 를 입력하면 makefile 의 전체 target 들이 수행될 수 있고, target 이름을 추가로 지정하면 해당 target 만 수행될 수 있음.

make 는 현재 디렉토리에 있는 makefile 을 읽어 작업을 수행함.

3) 종속 트리

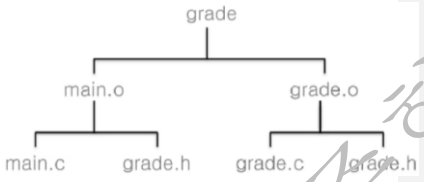
make 는 실행하면 makefile 을 읽어 종속 트리를 생성함.

make 는 종속 트리 상에서 상위 파일의 생성/수정 시간보다 바로 하위 파일의 생성/수정 시간이 최근인 경우 해당하는 target 의 명령만 수행함.

touch 로 파일 수정 시간을 새로 지정하여 make 작업 시 컴파일하도록 하기도 함.

메모 포함[이125]: 즉, dependencies에 target 파일의 생성/수정 이후에 생성/수정된 파일이 하나라도 존재하는 경우에만 그 target에 해당하는 명령을 수행한다. 이 과정의 작업 순서는 종속 트리로 이해할 수 있다.

4) 목적 파일로 효율적으로 컴파일하기




```
grade: main.o grade.o
	gcc -o grade main.o grade.o
main.o: main.c grade.h
	gcc -c main.c
grade.o: grade.c grade.h
	gcc -c grade.c
```

main.c 와 grade.c 를 따로 컴파일함.

5) make 의 편의 기능

1. .o 확장자를 가진 파일이 target 인 규칙에서는 dependencies 에 동일한 이름의 .c 파일을 생략할 수 있음.
-> .o 파일은 대응되는 .c 파일에 종속됨.

2. 내장된 여러 make 매크로 사용가능.
(ex.
main.o grade.o : grade.h
	gcc -c $*.c
인 경우, main 일 때는 별에 main 이, grade 일 때는 별에 grade 가 들어감.)

3. 매크로 선언
<매크로명> = <문자열> 형식으로 매크로를 선언할 수 있음.
문자열에는 따옴표를 씌우지 않음.
문자열의 끝에 ₩를 붙여서 다음 행에 이어서 작성할 수 있음.
$(<매크로명>) 형식으로 매크로를 사용할 수 있음.

4. make 파일에서 주석은 #으로 시작함.

5. make 의 명령은 실행 시 실행 내용을 출력하는데, @을 붙여서 출력되지 않게 할 수 있음.

3. gprof

GNU 프로파일러.

```
gprof <option> <file>
```

프로그램의 실행 profile 을 보여줌.

1) 동작 과정

1. -pg 옵션을 붙여 컴파일 한다.
2. -pg 옵션을 붙이면 각 함수의 호출 횟수와 실행 시간을 수집하는 코드를 실행 파일에 삽입한다.
3. -pg 옵션을 붙인 파일을 실행하면 실행 정보가 gmon.out 이라는 이름의 파일에 저장된다.
4. gprof 는 gmon.out 파일을 사용하여 실행 profile 을 생성/출력한다.

메모 포함[이126]: ex.

```
$ gcc -pg -static -o quicksort quicksort.c
$ quicksort
$ gprof -b quicksort
```

2) 인자 작성법

option.

-b : 간단한 profile 을 출력함.

file.

실행 profile 을 생성/출력할 실행 파일의 이름을 작성함.

3) profile 형식

실행 profile 은 플랫 profile 과 호출 그래프로 나뉨.

플랫 profile -> 각 함수의 호출 횟수와 실행 시간에 대한 내용.

호출 그래프 -> 각 함수들이 서로를 얼마나 호출하는지에 대한 내용.

-> name 부분에서 튀어나와 있는 숫자이 해당 부분 인덱스의 함수임.

-> 튀어나와 있는 부분 위에는 해당 함수를 호출한 함수들에 대한 정보가 있음.

-> 튀어나와 있는 부분 아래에는 해당 함수가 호출한 함수들에 대한 정보가 있음.

-> called 에서 + 는 재귀 호출을 의미함.

4. gdb

gprof <file>

대화식 디버거.
gdb 를 실행하면 gdb 프롬프트가 출력되며, 이곳에 명령을 입력하여 디버깅을 진행하게 됨.

gdb 를 사용하려면 컴파일 시에 -g 옵션을 사용해야 함.
이 경우, 컴파일 시에 디버깅을 위한 부가 정보가 실행 파일에 포함됨.

인자로는 디버깅을 진행할 실행 파일명을 작성함.

1) gdb 기능

run	-> gdb 에서 프로그램 실행.
list	-> 주변 소스 코드 보기. l(소문자 엘)로 줄여서 사용할 수도 있음.
	-> 인자에 출력할 코드의 구간을 지정할 수도 있음.
	-> 인자에 함수 이름을 지정할 수도 있음.
print <변수명>	-> 특정 변수의 값 출력.
set var <변수명> = <값>	-> 변수 값 지정.
break <함수명 or 행번호>	-> 검사점 설정.
	-> 함수명을 인자로 작성하면 해당 함수가 시작하는 행에 검사점을 설정함.
	-> 마지막에 조건을 명시하여 해당 조건이 참일 때 프로그램을 정지할 수 있음.
info break	-> 검사점 확인.
clear <행번호>	-> 검사점 삭제.
step	-> 프로그램을 한 행 씩 실행.
continue	-> 프로그램을 다음 검사점까지 실행.
backtrace	-> 함수 호출 관계 출력.
help	-> 도움말 기능.
	-> 뒤에 명령어 이름을 명시하여 해당 명령어에 대한 설명을 볼 수도 있음.
quit	-> gdb 종료.

메모 포함[이127]: ex.
list 13, 15
이런 형식으로 작성한다.

메모 포함[이128]: ex.
break 10 if j > 10

2) 검사점 사용 원리

검사점 설정 이후 프로그램을 실행하면 검사점 위치에서 프로그램이 정지함.
검사점은 필요한 위치에 멈춰서 코드나 값을 검사하기 위한 용도임.

프로그램 정지 이후 step 이나 continue 를 사용해서 실행 과정을 확인할 수 있음.

+ Segmentation fault

잘못된 메모리 공간에 접근하려고 시도하면 발생하는 오류 메시지.
주로 포인터나 배열에서 발생함.

6. 기타

1. main() 함수의 매개변수

프로그램을 실행 시에는 내부적으로 프로그램에 인자가 지정됨.
명령어 입력 시에도 인자가 지정됨. -> 이런 인자를 명령어 라인 인자라고 함.

c 에서 명령어 라인 인자는 main() 함수의 매개변수로 전달됨.
main() 함수 매개변수의 형과 기능은 이미 고정되어 있어서 프로그래머가 임의로 정할 수 없음.
변수의 식별자는 수정이 가능함.

main() 함수의 헤더 -> int main(int argc, char *argv [])
첫 번째 인자
-> 명령어라인 인자의 개수를 나타냄.
-> 프로그램명도 인자로 취급되기 때문에 이 값은 1 보다 크거나 같은 정수임.

두 번째 인자
-> 명령어 라인의 인자를 포인트함.
-> 명령어 라인의 인자들은 문자열로 전달되어 argv[0] ~ argv[argc-1]까지의 원소로 포인트됨. (포인터 배열)
-> argv[0]에는 프로그램명(첫 번째 단어)이 들어가므로 유의.
-> argv[argc]에는 NULL 이 저장되어 있음.

+ 우분투에서 main()으로의 인자 전달 형식

./a.out <인자 1> <인자 2> <인자 3> . . . 의 형식으로 작성함.