

(8/2) DAY01

@ 4차 산업 혁명이란? 키워드

- 1차 산업혁명 : 증기기관의 발명 (기계화)
- 2차 : 대량생산가능 (컨베이어벨트로 인한 자동차 산업 등에서 소비의 대중화가 시작됨)
- 3차 : 컴퓨터의 발달 -> 정보화 사회
- 4차 : 인공지능(AI)
A B C ? => AI / Big data / Cloud

단기프로젝트라도 나를 표현하기 위한 제목을 잘 정해야한다!

@ computer programming?

- CPU(Central Processing Unit, 중앙처리장치)가 이해할 수 있는 명령어를 작성하는 것
- CPU는 이진수만 이해가능
ex) 83 C0 01 (16진수 표현, intel x86 mnemonic)
=> add eax, 1 => 다른 말로 assembly 언어 (low level 언어)
=> i = i + 1; or i++; (i변수에 1을 더해서 i에 저장) : 우리가 작성해야할 것

@ programming 카테고리

1. 변수 : 어떤 데이터가 저장되는
 2. 연산자 : 사칙연산/논리연산/비트연산 등
 3. 조건문 - 분기문
 4. 반복문
 5. 함수(function) = 서브루틴(subroutine) = module
- 대부분의 언어에서
이 구조들은
거의 똑같음



-
6. 라이브러리 (동적/정적) : 필수요소의 집합, 매뉴얼 참조
 7. Preprocess(전처리) / Header
 8. 개발 / 실행 환경

@ 프로그래밍 수행방식

1. compiler : 번역기가 실행파일 생성 ex) C/C++, C#
 2. interpreter : 실시간 번역기 ex) BASIC, PYTHON
- ※ JAVA는 컴파일러와 인터프리터의 중간 정도라고 이해하면 됨

(8/3) DAY02

@ 프로그래밍 작성 및 실행 순서

1. 프로그램 작성 : 에디터에서 작업
2. 컴파일
3. 링크
4. 실행파일 생성

@ IDE?

- Integrated Development Environment : 합 개발 환경
- **editor** + compiler + linker + **debugger** = integrated

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

#include : preprocess = header 파일

<stdio.h> : 파일(file)명

: 함수

=> 컴파일 후, 커맨드창(=CMD, consloe)에 결과가 출력됨

```
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
}
```

iostream : 파일명(=문자열, 가변적이지 않음)

@ 함수에 대한 이해

- 적절한 입력과 그에 따른 출력이 존재 하는 것을 가리켜 함수라고 함
- C 언어의 기본 단위는 함수
 $y = f(x)$ => 입력 x / 출력 y
C언어에선 입력은 인수, factor, argument 등으로 칭함

@ 헤더 파일의 이해

- stdio.h = std(standard, 표준) + io(input & output, 입출력) + .h(헤더파일)
- stdlib.h = std + lib(library, 라이브러리)
- conio.h = con(console, 콘솔) + io

@ 주석 : // or /* */

@ 간단한 코드 해석

```
#include <stdio.h>
#include<conio.h>
int main()
{
    //const char* name = "혜빈";
    char name[10] = "혜빈";
    int age = 10;
    printf("안녕하세요. 반갑습니다!\n");
    printf("저는 %d살 입니다.", age);
    getch();

    return 0;
}
```

- %d : 10진 정수 데이터를 변환하여 출력
d(decimal)
- getch() : 콘솔 입력 함수
- C++에서의 문자열은 이와 같이 표현!

@ 데이터 변환 서식 문자

서식 문자	출력 형태
%c	단일 문자 ex) 'a', 'b', 'c', '1'
%s	문자열 ex) "abc"
%d	부호 있는 10진 정수
%f	부호 있는 10진 실수
%x	부호 없는 16진수

@ 데이터 타입 별 저장 공간

데이터 타입	필요한 저장 공간	표현 가능한 범위
char	1 byte	
int	4 byte (= 32 bit)	±2G
float	4 byte	±2G
double	8 byte	±4G

(8/4) DAY03

2강. 변수와 연산자

@ 변수? 데이터를 저장할 수 있는 메모리 공간

@ 다양한 형태(자료형)의 변수

- 정수형 : char(단일문자형), int, long
- 실수형 : float, double

@ 변수 선언 시 주의 사항 (함수도 동일함)

- 변수의 이름은 알파벳, 숫자, 언더바(_)로 구성
- 대소문자 구분 (SQL은 구분안함)
- 변수 이름은 숫자로 시작 불가, 알파벳과 언더바로 시작가능
- 공백 포함 불가

@ 변수와 다른 상수!

- 상수도 메모리 공간을 할당받음, 하지만 데이터의 변경은 불가능!
- 프로그램 소스에 삽입되어 있는 모든 숫자, 문자열

@ 대입연산자와 산술연산자

- =, +, -, *, /, %, &, |, !

@ 논리연산자 : &&(and, 논리곱), ||(or, 논리합), !(not)

@ 자료형 변환

- 강제 형 변환(cast 연산)

(8/5) DAY04

알고리즘 구현

3강. 분기문/ 반복문

@ 반복문의 기능 : 특정 영역을 특정 조건이 만족하는 동안에 반복 실행하기 위한 문장

@ 세가지 형태의 반복문

- while / do~while / for

@ 프로그래머가 반복문 구성을 위해서 체크해야할 부분

1. index (= while문의 수행 횟수)
2. 수행조건
3. index의 증감

@ while문

```
while(1) // while(조건식)
{
    반복내용 코드
}
```

- 조건식이 참(true)이면 while문 안의 코드가 반복됨

- 옆의 예시와 같이 조건식이 항상 참인 경우는 무한루프라 칭함

@ for문

@ 구구단 프로그램 작성

(예시1) 강사님 예시 -> 복잡도가 N^3

```
#include <iostream>
int main()
{
    int i, j, k;

    // N X M 구구단 프로그램
    for (i = 2; i < 10; i += 4) // N
    {
        for (j = 1; j <= 9; j++) // M
        {
            for (k = 0; k < 4; k++)
            {
                printf("%2d * %2d = %3d  ", i + k, j, (i + k) * j);
            }
            printf("\n");
        }
        printf("\n");
    }
}
```

(예시2) 내가 작성한 것 -> 복잡도 N^2

```
#include <iostream>
int main()
{
    int i, j;
    // N X M 구구단 프로그램
    for (i = 1; i <= 9; i++) // 구구단의 M
    {
        for (j = 2; j <= 5; j++) // 구구단의 N (2 ~ 5단)
        {
            printf("%2d * %2d = %3d", j, i, i * j); // j = N, i = M
        }
        printf("\n");
    }
    printf("\n");
    for (i = 1; i <= 9; i++) // 구구단의 M
    {
        for (j = 6; j <= 9; j++) // 구구단의 N (6 ~ 9단)
        {
            printf("%2d * %2d = %3d", j, i, i * j); // j = N, i = M
        }
        printf("\n");
    }
}
```

8강. 조건문

@ if문에 의한 조건적 실행

```
if( 실행 조건 )
{
    실행하고자 하는 내용
}
else if( 실행 조건 )
{
    실행하고자 하는 내용
}
else
{
    실행하고자 하는 내용
}
```

- 실행조건이 만족되는 경우 코드문 진행

- if문의 단점 : loss가 발생 (=불필요한 지연)

@ break / continue ★★

- break : 반복문을 빠져 나올 때 사용

=> for, while, do-while, switch 문에서 사용!

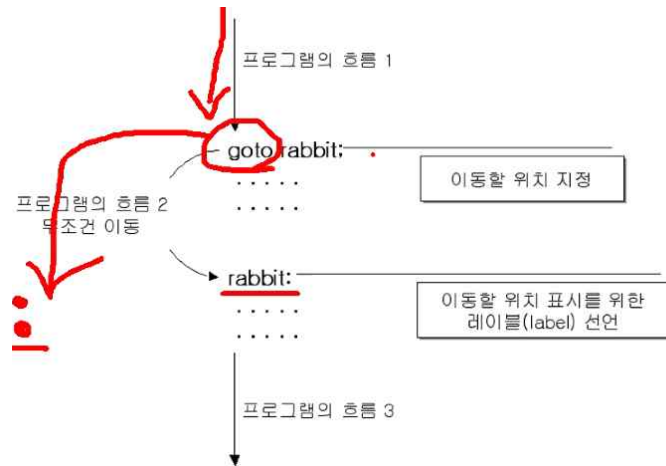
- continue : 아래의 코드 실행 X

다음 번 반복으로 넘어갈 때 사용 (다시 반복으로 돌아감)

=> 반복문에서 사용, switch문에서는 사용하지 않음

@ GOTO label문

- 구조적인 언어 (ex, C언어 등)에서는 가급적 사용하지 말자!



@@ 블록 주석처리 하기 : ctrl + k + c

@@ 블록 주석처리 해제 : ctrl + k + u

@ switch문

- 조건식은 반드시 정수 값이어야 한다.
- int, long, char 가능 / float, double, 문자열 불가능

※ switch 문의 case문은 GOTO label과 같음

-> 따라서 break;를 써주지 않으면 모든 case문을 방문하기 때문에 꼭 break를 사용해주자!

```
switch( 조건식 )
{
    case 1:
    {
        실행영역
        break;
    }
    case 2:
    {
        실행영역
        break;
    }
    default:
    {
        실행영역
        break;
    }
}
```

@ 조건문을 사용하여 다음 조건을 만족시키는 프로그램을 작성하라.

- 1~9까지의 숫자를 입력받아 영어로 출력, 0은 종료

```
#include <stdio.h>
int main()
{
    int a;
    while (1)
    {
        scanf_s("%d", &a);
        if (a == 1) printf("One\n");
        else if (a == 2) printf("Two\n");
        else if (a == 3) printf("Three\n");
        else if (a == 4) printf("Four\n");
        else if (a == 5) printf("Five\n");
        else if (a == 6) printf("Six\n");
        else if (a == 7) printf("Seven\n");
        else if (a == 8) printf("Eight\n");
        else if (a == 9) printf("Nine\n");
        else if (a == 0) break;
    }
}
```


@ (심화) 조건문을 사용하여 다음 조건을 만족시키는 프로그램을 작성하라.

- 0~9까지의 숫자를 입력받아 영어로 출력
- 이외의 문자 또는 숫자를 입력받으면 프로그램 종료

```
#include <stdio.h>

int main()
{
    char a;
    while (1)
    {
        // scanf는 엔터가 입력되는 순간까지의 값을 버퍼에 저장함
        scanf_s("%c", &a);
        // getchar() : 호출될때마다 버퍼 값을 하나씩 지움
        // 엔터값(LF)을 만날 때까지 입력버퍼를 비워주는 역할
        while (getchar() != '\n');

        // switch문 아스키코드값 사용
        switch (a)
        {
            case 48: printf("Zero\n"); break; // 48 = '0'의 아스키코드
            case 49: printf("One\n"); break;
            case 50: printf("Two\n"); break;
            case 51: printf("Three\n"); break;
            case 52: printf("Four\n"); break;
            case 53: printf("Five\n"); break;
            case 54: printf("Six\n"); break;
            case 55: printf("Seven\n"); break;
            case 56: printf("Eight\n"); break;
            case 57: printf("Nine\n"); break;
            default: break;
        }
        if ((a > 57) || (a < 48)) break; // '0' ~ '9' 이외가 입력될 경우
    }
    return 0;
}
```

- while(getchar() != '\n') 가 생략된 경우, 처음 입력된 문자 이후에 '\n'이 버퍼에 남아있기 때문에 다음 루프에서 a가 '\n'이 입력되어 루프를 벗어나게 됨!

(8/6) DAY05

4강. 함수와 배열

@ main 함수 다시 보기 : 함수의 기본 형태

```
int main(void)
{
    함수의 몸체
}
```

- 반환의 형태 : int
- 함수의 이름 : main
- 입력의 형태 : (void)

@ 전역변수 : 변수명은 가능한 길고 명확하게 설정하자 (지역변수와 오버로드 발생 가능성 있음)

- 프로그램이 어디에서나 접근이 가능한 변수

10강. 1차원 배열

@ 배열?

- 둘 이상의 변수를 동시에 선언하는 효과 (단, 동일한 데이터 타입의 변수!)
- 배열 요소의 위치 표현 : 인덱스(index)는 0에서부터 시작한다!

(8/09) DAY06

10강. 1차원 배열

@ 배열 선언에 필요한 3가지

- 배열 요소 자료형 : 배열을 구성하는 변수의 자료형
- 배열 이름 : 배열에 접근할 때 사용되는 이름
- 배열 길이 : 배열을 구성하는 변수 개수 (반드시 상수 사용) ←

int array[10];

=> **const**를 사용하여 변수를 상수 처리해 배열 길이를 수정할 수 있음

```
const int student = 10;
int main(void)
{
    // int array[10];
    int array[student];
}
```

@ 문자열(=문자의 배열) : 상수 & 변수

@ 문자열과 char형 배열의 차이점

char arr1[] = "abc";	arr1 = a b c \0 : 길이 4
char arr2[] = {'a', 'b', 'c'};	arr2 = a b c : 길이 3
char arr3[] = {'a', 'b', 'c', '\0'};	arr3 = a b c \0 : 길이 4

11강. 다차원 배열

@ 1차원 배열 / 다차원 배열

- 1차원 배열 : 인덱스가 1개 / 다차원 배열 : 인덱스가 여러 개

@ 다차원 배열의 실제 메모리 구성

- 1차원 배열과 동일하지만, 접근 방법을 2차원적으로 해석할 뿐! 하지만 2차원적으로 이해할 것

@ 2차원 배열 초기화

```
int array1[][] = {1,2,3,4,5,6,7,8,9}; // Error!
int array2[][3] = {1,2,3,4,5,6,7,8,9}; // OK
int array3[][4] = {1,2,3,4,5,6,7,8,9}; // OK
```

(8/10) DAY07

12강. 다차원 배열과 포인터(*)

@ 포인터 : 주소를 가리키는 변수

<pre> int main() { int arr1[10]; function1(arr1); // OK } void function1(int *a) { ... } </pre>	<pre> int main() { int arr2[10][10]; function2(arr2); // Warning! } void function2(int **a) { ... } </pre>
--	---

- 예제1) 1차원 배열과 포인터

arr1 : (int)

0000	1	0004	2	0008	3	0012	4
------	---	------	---	------	---	------	---

int는 4byte므로 배열의 한 공간씩을 4byte를 차지함

arr1[2] = *(a + 2) => * == []
 ≠ *(a + 8)

=> 포인터가 주소값을 가리키지만, 포인터 표현시 주소값 만큼 증가 시키는 것이 아님!

- 예제2) 2차원 배열과 포인터

arr2 :

0001	1	0002	2	0003	3	0004	4
0005	5	0006	6	0007	7	0008	8
0009	9	0010	a	0011	b	0012	c

arr2[1][2] = * (* (a + 1) + 2)

@ 포인터 사용방법

- 변수의 주소 값 : &a
- 포인터를 사용해 주소 값의 데이터 값을 참조 : *a
- swap() 예제로 이해하는 포인터 사용법

```
void swap(int* a, int* b) // a,b를 포인터로 선언하고 전달된 매개변수 값으로 설정(초기화)
{
    // 포인터 사용방법 - 포인터가 가리키는 주소의 값 : *p
    // - 주소 자체 : p
    // 여기서 a와 b는 주소 값을 주의하자!
    int temp;
    printf(" input> a(%08x) : %d, b(%08x) : %d\n", a, *a, b, *b);
    temp = *a; // *a 포인터가 가리키는 주소의 값을 가져오기 위함
    //a = b; warning! (a의 주소값이 b를 가리키게 되어 b값이 바뀌면 a도 동기화
    //됨)
    *a = *b;
    *b = temp;
    printf(" result> a(%08x) : %d, b(%08x) : %d\n", a, *a, b, *b);
}

int SwapTest()
{
    int a = 50, b = 60;

    // %x : 16진수 출력, %8x : 8자리로 16진수 출력, %08x : 빈자리는 0으로 채움
    printf("original> a(%08x) : %d, b(%08x) : %d\n", &a ,a, &b, b);

    //swap a & b
    //swap(a, b); warning! (a, b가 변하지 않음)
    swap(&a, &b); //주소값 전달을 위해 &를 붙여줌

    printf("after swap> a(%08x) : %d, b(%08x) : %d\n", &a ,a, &b, b);

    return 0;
}
```

- 결과

```
original> a(00b4fdac) : 50, b(00b4fda0) : 60
input> a(00b4fdac) : 50, b(00b4fda0) : 60
result> a(00b4fdac) : 60, b(00b4fda0) : 50
after swap> a(00b4fdac) : 60, b(00b4fda0) : 50
```

(8/11) DAY08

@ 포인터 배열의 예

```
const char* arr[] = {"A", "B", "C"};
```

- *[] == [][] == ** : string * (스트링 배열)
- char* = string

@ 예시1

```
#include <stdio.h>

int main()
{
    int a = 10, b = 20, c = 30;
    int* arr[3] = {&a, &b, &c}; // arr는 a,b,c의 주소값을 갖는 배열

    printf("%d\n", *arr[0]);
    printf("%d\n", *arr[1]);
    printf("%d\n", *arr[2]);

    return 0;
}
```

@ 예시2

```
#include <stdio.h>

int main()
{
    const char* arr[3] = {"Apple", "Banana", "Carrot"};
    printf("%s\n", arr[0]);
    printf("%s\n", arr[1]);
    printf("%s\n", arr[2]);

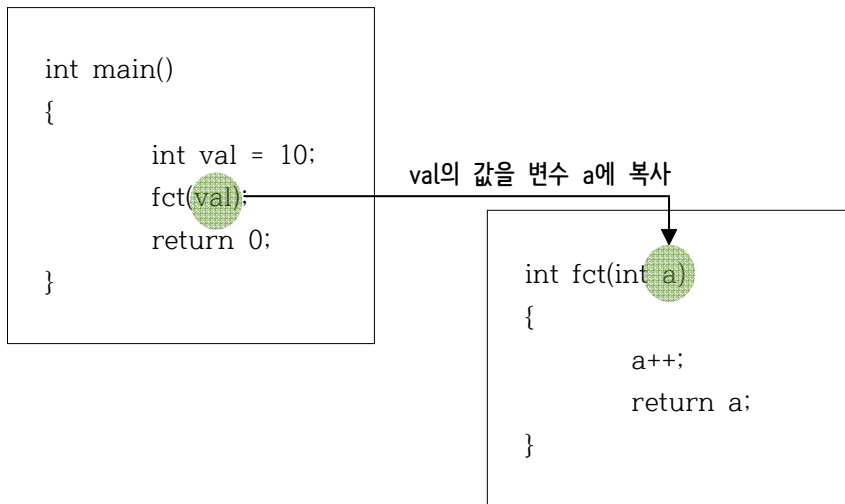
    return 0;
}
```

@ SortTest.cpp에서 string형 name2 스왑을 위한 swapEX2() 이해하기

15강. 포인터와 함수에 대한 이해

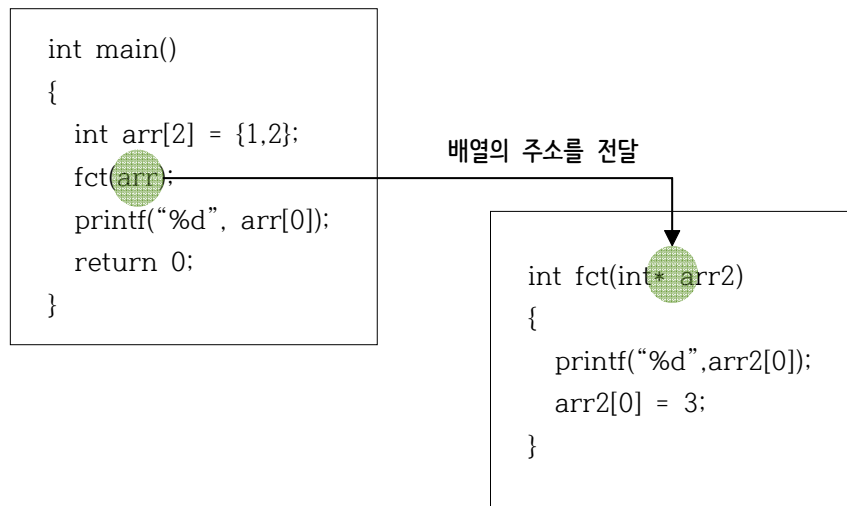
@ 기본적인 인자의 전달 방식

- 값의 복사에 의한 전달(call-by-value)



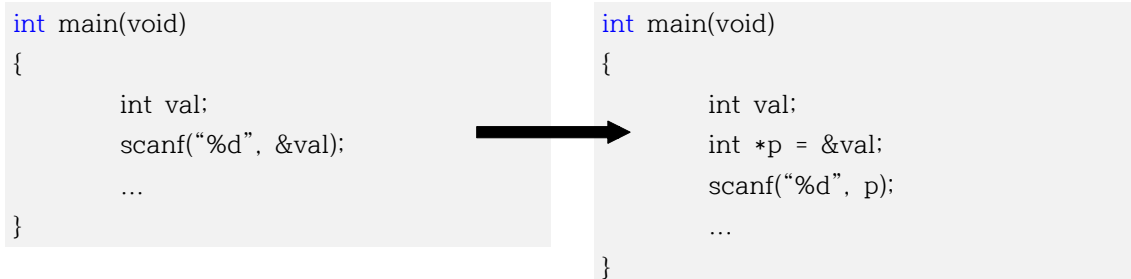
@ 배열의 함수 인자 전달 방식

- 배열 이름(배열 주소, 포인터)에 의한 전달 (참조에 의한 전달, call-by-reference)



@ scanf 함수 호출 시 &를 붙이는 이유

- 주소 전달을 위해서!



@ 포인터와 const 키워드 : 문자열을 위한 const 이외에는 사용하지 말자

- 포인터가 가리키는 변수의 상수화 : arr는 상수 string

```
const char* arr[] = {"A", "B", "C"};
```

```
int a = 10;
```

```
const int *p = &a;
```

```
*p = 30 // Error!
```

```
a = 30 // OK!
```

- p는 상수 주소값
- 초기화 후 변경 불가
- 따라서 한번 정해지면 바뀌지 않음

- const 키워드를 사용하는 이유
 - 컴파일 시 잘못된 연산에 대한 에러 메시지
 - 프로그램을 안정적으로 구성

16강. 포인터와 포인터

@ 포인터의 포인터 : 더블 포인터

- 문자열 배열(string 배열), 2차원 배열 등에서 사용됨
- 구현 사례 : 더블포인터 입장에서의 swap

X : 주소값을 바꾼 것. 값이 바뀌지 않음	OK : 올바른 swap
<pre>void swapEx2(const char** a, const char** b) // 문자열 swap : string * (스트링 포인터) { const char **temp; temp = a; a = b; b = temp; }</pre>	<pre>void swapEx2(const char** a, const char** b) // 문자열 swap : string * (스트링 포인터) { const char *temp; temp = *a; *a = *b; *b = temp; }</pre>

17강. 함수포인터와 void포인터

@ 함수이름의 포인터 타입을 결정짓는 요소 : function pointer <-> SDK (간단한 이해정도만 하자)

- 리턴 타입 + 매개변수 타입

```
int fct1(int a)
```

```
{
```

```
    a++;
```

```
    return a;
```

```
}
```

```
int (*fPtr1)(int);
```

```
fPtr1 = fct1; // function pointer
```


```
fPtr1(10);
```


@ void포인터 : type이 없는 포인터

- 자료형에 대한 정보가 제외된, 주소 정보를 담을 수 있는 형태의 변수
- 포인터 연산, 메모리 참조와 관련된 일에 활용할 수 없다. 값을 바꿀 수 없음
- 임시로 선언해놓고 => casting(강제 형 변환)을 수행하여 사용

```
int main()
{
    char c = 'a';
    int n = 10;
    void* vp;
    vp = &c;
    vp = &n;
    ...
}
```

```
int main()
{
    int n = 10;
    void* vp = &n;
    *vp = 20;           // Error!
    vp++;               // Error!
    ...
}
```



- ArrayTest.cpp -> VoidTest()

```
void VoidPrint(void* p, int i) // argument : void pointer
{
    if (i == 1)
    {
        char* cp = (char*)p; // char형으로 casting
        printf("%c\n", *cp);  // cp = p의 주소이므로 값을 출력하려면
                               // *cp 여야 올바른 값 출력
    }
    else if (i == 2)
        printf("%d\n", *(int*)p); // int형으로 casting
    else if (i == 3)
        printf("%f\n", *(double*)p); // double형으로 casting
}

void VoidTest()
{
    char c = 'z';
    int n = 10;
    double a = 1.414;

    // void 포인터
    void* vp;
    VoidPrint(vp = &c, 1);
    VoidPrint(vp = &n, 2);
    VoidPrint(vp = &a, 3);
}
```

@ void pointer를 사용하여 SortTest.cpp의 swap함수 합치기

- SortTest.cpp에서 자료형에 따라 swap함수를 여러 개 선언한 것을 하나로 합치기

ex) swapEx(int), swapEx1(double), swapEx2(const char*)

=> AllSwap(void* a, void* b, int i) : i는 자료형을 구분하기 위한 인자

```
void AllSwap(void* a, void* b, int i)
{
    int itemp; double dtemp; char ctemp;
    if (i == 1) // 1바이트 : char
    {
        ctemp = *(char*)a;
        *(char*)a = *(char*)b;
        *(char*)b = ctemp;
    }
    else if (i == 4) // 4바이트 : int, float, const char*(string)
    {
        itemp = *(int*)a;
        *(int*)a = *(int*)b;
        *(int*)b = itemp;
    }
    else if (i == 8) // 8바이트 : double
    {
        dtemp = *(double*)a;
        *(double*)a = *(double*)b;
        *(double*)b = dtemp;
    }
}
```

22강. 구조체와 사용자 정의 자료형

@ 구조체 정의

```
키워드      구조체 이름
struct point // point라는 이름의 구조체 선언
{
    int x;    // 구조체 멤버 int x
    int y;    // 구조체 멤버 int y
};
```

@ 구조체 변수의 접근(access)

```
struct point {  
    int x;  
    int y;  
};  
  
int main()  
{  
    struct point p1;  
    p1.x = 10;  
    p1.y = 20;  
    ...  
    return 0;  
}
```

@ 구조체 변수의 초기화

- 배열 초기화 문법과 일치
- 그럼 배열이랑 구조체 차이점은?
 - > 배열은 동일한 data type의 집합 / 구조체는 동일하지 않은 data type의 집합

(8/12) DAY09

@ 중첩된 구조체

- 구조체의 멤버로 구조체 변수가 오는 경우

```
struct point {
    int x;
    int y;
};

struct circle {
    struct point p;
    double radius;
};

int main()
{
    //struct circle c = {1, 2, 3.0};
    struct circle c = { (1, 2), 3.0};
    //두 개는 동일하지만, 아래의 경우로 쓰길 권장
}
```

@ 자료형의 이름을 새롭게 지어주기 위한 키워드 : **typedef**

- typedef int INT; // INT라는 이름을 기본 자료형 int에게 이름을 지어준다

```
struct point {
    int x;
    int y;
};
typedef struct point point;
```

case 1

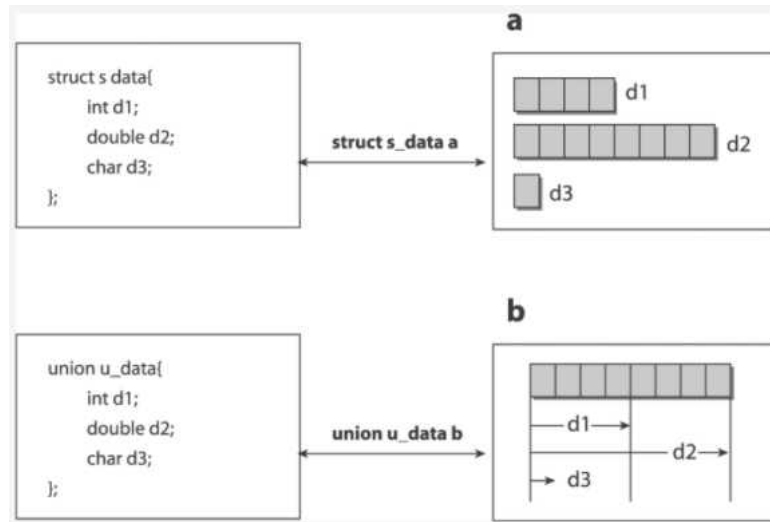
```
typedef struct point{
    int x;
    int y;
} point;
```

case 2 (일반적)

- case1과 case2는 동일함! case1의 경우는 다른 프로그래머의 구조체를 가져다 사용할 때 쓰이는 경우
- case2의 *point*는 대게 생략해서 사용
 - => 생략하지 않은 경우에는 본문에서 struct point가 사용가능함 (근데 보통은 사용하지 않음)

@union - 공용체 (C언어에서만 사용하는 키워드)

- 공용체의 특성 : 하나의 메모리 공간을 둘 이상의 변수가 공유하는 형태
- 주로 통신 영역에서 고속처리를 위해 사용하는 경향이 있음 -> 바이트 단위로 송수신



- 일반적으로는 위의 그림과 같이 사용되지는 않고 아래와 같이 사용됨

```
union data{
    char arr[8];
    int arr1[2];
    double a;
}
```

-> 8개의 char형태로 받아서 따로 변환 없이 바로 int나 double로 데이터를 처리하는 등에 사용됨

@ enum (enumerative) - 열거형

- 열거형의 정의와 의미 : 정수의 값에 이름을 붙여준다고 생각하자!

```
enum color {RED = 1, GREEN = 3, BLUE = 5};
```

상수 RED(1), GREEN(3), BLUE(5)의 선언
color라는 이름의 자료형 선언

- 열거형 사용 이유
- 특정 정수 값에 의미 부여 가능 / 프로그램의 가독성을 높이는데 도움이 됨

(8/13) DAY10

7강. C 표준 함수

- C 표준 함수는 모든 플랫폼(H/W : PC, Mac, 라즈베리 등 / OS : 윈도우, 맥OS, 리눅스 등)에서 사용 가능

@ 문자와 문자열 처리 함수

- strlen() : 문자열 길이 반환 함수 / strcpy() : 문자열 복사하는 함수 => 표준함수들

@ 입출력의 이해 : 콘솔 - 키보드 입력/모니터출력, 파일, 소켓 입출력

- 파일 : 텍스트 파일 / binary 파일 => 2가지 종류의 파일이 존재함
- 콘솔 : 파일의 특수한 형태
- 소켓 : 네트워크(특히 이더넷) 통신 프로그램을 위한 통신 디바이스

@ 표준 입출력 스트림 => File 개념

- 프로그램 실행 시 자동으로 생성 및 소멸
- 모니터와 키보드를 대상으로 함
- stdin, stdout, stderr => 표준 입출력 파일 => CONSOLE (std : standard)

- 문자 출력 함수 : putchar() : 단일문자 출력, fputc() : 파일 출력
 - 문자 입력 함수 : getchar() : 단일문자 입력, fgetc() : 파일 입력
- 잘 사용하지는 않음

@ EOF(End Of File)에 대한 이해

- fgetc, getchar 함수가 파일의 끝에 도달하는 경우 반환
- 파일을 읽을 때, 파일의 끝에 도달했다는 read 함수의 return 값을 표현하기 위한 상수 (-1값을 지님)
=> 기록을 할 때에도 마찬가지

@ fgets(buf, 1024, stdin)

- stdin, stdout -> 이미 존재하면서 언제라도 사용가능함 (즉, 다른 파일들은 사용 준비가 되어있지 않음)

```
void StreamTest()
{
    char buf[1024];
    FILE* f = fopen("C:\\Users\\hallo\\aa", "r"); //파일 이름은 포인터로 정의
    // "r" : 입력용, "w" : 출력용(overwrite : 기존 내용을 지우고 새롭게 작성)
    // "a" :
    fgets(buf, 1024, stdin);
    fputs("==== 입력문자열 ====> ", stdout);
    fputs(buf, stdout);
}
```

(8/18) DAY11

7강. C 표준 함수

- 입·출력의 이해 : 파일, 콘솔(=특수파일), 소켓 입·출력
- 스트림에 대한 이해 : 데이터를 송·수신하기 위한 일종의 다리

@

- stdout : 모니터에 출력하기 위한 키워드
- 파일 명의 옵션이 “w”, “a”일 경우 -> 저장되는 파일은 Windows(CRLF)로 자동 변환되어 저장됨
- “a+b” (바이너리 모드)일 경우 -> 저장되는 파일은 Unix(LF)

@ 파일 입력 함수

- fgets() : 문자열, *fgetc()* : 단일문자, fscanf() : 형식 지정 입력 함수
- fscanf(file pointer, “%d”, &변수명)

@ 파일 출력 함수

- fputs(), fputc(), fprintf()

(8/19) DAY12

@ 문자열 함수

- strcpy() : 문자열 복사 함수

```
int kor[] = {67, 70, 77, ...};
int eng[] = {70, 75, 80, ...};
char name[] = {"홍길동", "홍길이", "홍길삼", ...};

...

int main()
{
    ...
    for(int i=0; i<num; i++)
    {
        student[i].kor = kor[i];
        student[i].eng = eng[i];
        strcpy(student[i].name, name[i]);
    }
}
```

- 문자열은 대입 연산자를 사용할 수 없기 때문에 문자열 복사 함수 **strcpy()**를 사용해야한다!
 - strncpy(char* dest, const char* src, size_t n) : '0'이 포함되지 않음
 - size_t : Type define (long이라 이해하자)
- ex) strncpy()

```
char* a = "Good morning";
char buf[10];

strncpy(buf, a, 4);
printf("%s", buf);    // 결과 : Good?????? (쓰레기 값이 포함되어 출력)
                     // 즉, strncpy() 함수는 뒤에 null 값이 포함되지 않음

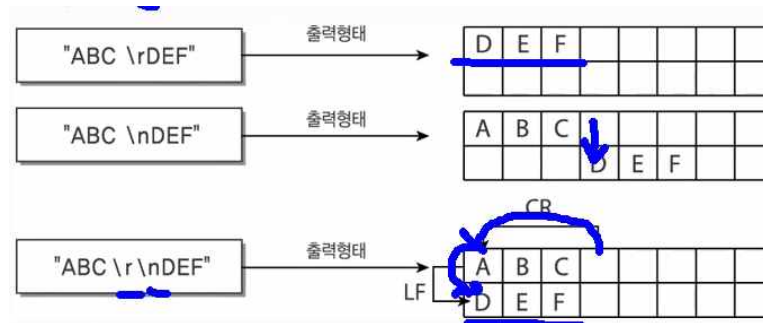
buf[4] = NULL;
printf("%s", buf);    // 결과 : Good
```


- 문자열 결합 함수 : `strcat()`, `strncat()`
- 문자열 비교 함수 : `int strcmp()` => 조건문에서 자주 사용됨
 - `strcmp(str1, str2)`는 -1, 0, 1 을 반환함
 - `str1 < str2 => -1` 반환 / `str1 == str2 => 0` 반환 / `str1 > str2 => 1` 반환
- 문자열을 숫자로 변환하는 함수 : `atoi()`, `atol()`, `atof()` ==> `<stdlib.h>`
 - `int atoi(char* ptr);` // ascii to integer, 문자열을 int형으로 데이터 변환
 - `long atol(char* ptr);` // ascii to long, 문자열을 long형으로 데이터 변환
 - `double atof(char* ptr);` // ascii to float, 문자열을 double형으로 데이터 변환
- 대소문자의 변환을 처리하는 함수들 : `toupper()`, `tolower()` ==> `<ctype.h>`
 - `int toupper(int c);` //소문자를 대문자로
 - `int tolower(int c);` //대문자를 소문자로

25강. 파일 입출력

@ CR & LF

- CR(Wr) : 앞으로 이동 / LF(Wn) : 다음 줄로 이동
- C언어는 Unix 모드이기 때문에 LF만 사용해도 CR이 자동으로 붙음



@ `fopen()` 후 에는 `fclose(file pointer)`를 꼭 해주자 (or `fcloseall()` : argument 필요 없음)

@ Windows 개발 환경에서는 콘솔 입출력 함수는 사용하지 않음 (파일입출력사용)

@ Random Access : 컴퓨터 기록장치, DB

- 특정 위치로 임의 접근 방식의 입·출력

26강. 메모리 관리와 동적 할당

@ 스택, 힙 그리고 데이터 영역

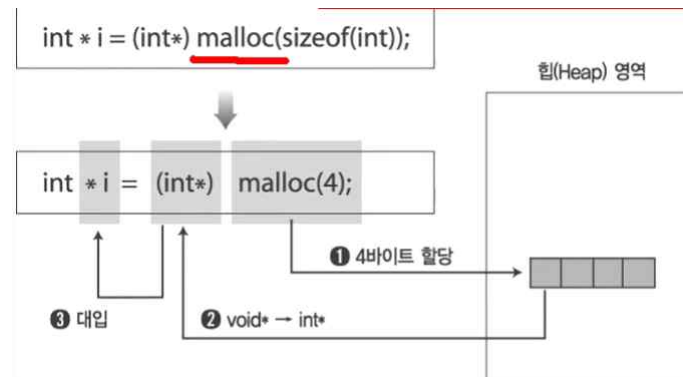
- 프로그램의 실행을 위해 기본적으로 할당하는 메모리 공간
 - 컴파일 타임에 함수에서 요구하는 스택의 크기 결정되어야함
 - 데이터 영역 : 전역변수, static 변수 (고정된 영역, 컴파일러가 정해주기 때문)
 - 힙(heap) 영역 : 프로그래머 할당
 - 스택(stack) 영역 : 지역변수, 매개변수
- > 가변 영역

@ 메모리 동적 할당 => malloc : memory + allocation

- 런 타임에 메모리 공간의 크기를 결정지어서 할당 (힙 영역에 할당)

@ 동적 할당된 메모리 공간의 소멸

@ malloc 함수의 활용 => void*(포인터 연산, value 참조 불가 => 따라서 cast필요-❷) 반환하는 메모리 확보 함수



- `int i[sizeof(int)];` //배열을 동적으로 선언한 것과 같다고 생각하자
 - 배열의 크기는 상수만 가능하지만, malloc 함수를 사용해 동적으로 선언이 가능해짐
- ex) 크기 7인 student 자료형(구조체)의 배열을 동적 할당하라 (GradeProcessing.cpp)

```
int main()
{
    int num;
    student* Students;    // malloc를 통해 메모리 확보
    ...
    fscanf(fftable, "%d", &num); // 파일입출력을 통해 num=7 read
    Students = (student*)malloc(sizeof(student) * num);
    ...
}
```

- ※ `student* Students = (student*)malloc(sizeof(student) * num);`
- ※ `(student*)` : malloc의 반환형이 `void*` 이므로 `(student*)`형태로 형 변환 해달라는 것
- ※ `sizeof(studnet)` : `sizeof()`는 괄호 안에 자료형 타입을 바이트로 연산해주는 연산자
- ※ `sizeof(studnet)*num` : `Students[num]`과 동일한 크기의 메모리 할당을 위한 것

@ # 으로 시작하는 전처리기 지시자

- `#include <>` : 뒤에 나오는 시스템 헤더<>를 포함시키는 지시자
 - 프로그래머가 만든 헤더를 사용할 때는 “”를 사용
- `#define PI(매크로) 3.1415(대체리스트)`
 - 컴파일러에 의해 처리되는 것이 아님
 - 전처리기에겐 단순 치환 작업을 요청할 때 사용되는 지시자 (=변수 선언과 비슷)

@ 매크로 함수란?

- 매크로를 기반으로 정의되는 함수
- 함수가 아니라 매크로다!! 다만 함수의 특성을 지닐 뿐
 - `#define SQUARE(x) x*x` // x : 매크로함수 인자
- 자료형에 독립적이며 실행속도가 향상
- 구현이 어렵고 디버깅이 어려운 단점

28강. 모듈화 프로그래밍

@ 모듈(module)이란 무엇인가? (의미적인 것 설명)

- 프로그램을 구성하는 구성 요소의 일부
- 관련 데이터와 함수들이 묶여서 모듈을 형성
- 파일 단위로 나뉘는 것이 보통

@ 모듈화 프로그래밍

- 기능별 파일을 나뉘가며(= 작업을 나뉘서) 수행
- 유지 보수성 좋아짐

@ 파일의 분할 및 컴파일

- 파일을 나눌지라도 완전히 독립되는 것은 아님
- 파일이 나뉘어도 상호 참조 발생 가능 -> 이는 전역 변수 및 전역 함수로 제한됨

- **extern** 키워드 : 변수, 함수 등 전역으로 다른 소스 파일에서도 사용할 수 있도록 해주는 키워드

@ `#if`, `#elif`, `#else`, `#endif` 기반 조건부 컴파일

@ 헤더 파일 포함 관계에서 발생하는 문제

- 하나의 헤더 파일을 두 번 이상 포함!

(8/20) DAY13 - C++

* C++ 특징 이해하기

@ 함수 오버로딩의 이해

- 매개 변수의 선언(자료형, 개수)이 다르다면 동일한 이름의 함수를 정의할 수 있음 => 함수 오버로딩
- 매개변수가 동일하면서 return이 다른 경우는 오버로딩 되지 않음

@ 매개변수의 디폴트 값

- 인자를 전달하지 않은 경우, 디폴트 값으로 저장 됨
- 디폴트 값을 지정할 때는, 뒤에서부터 설정해야한다!

ex) `int YourFunc(int num1, int num2, int num3 = 10) { ... }` (o)

`int YourFunc(int num1 = 50, int num2, int num3) { ... }` (x)

@ namespace의 기본원리

- 프로젝트의 모듈화를 위한 것, 전역 변수와 함수가 포함된 큰 범위의 모듈

(8/23) DAY14 - C++

* Reference

@ reference의 이해

- `int num1 = 10;` => 변수의 선언으로 `num1`이라는 이름의 메모리 공간이 할당
- `int &num2 = num1;` => reference의 선언으로 `num1`의 메모리 공간에 `num2`라는 이름이 추가로 붙음
- reference는 기존에 선언된 변수에 붙이는 '별칭', 변수 이름과 별 차이 없음

@ return 값이 reference인 경우

- `int& RefRetFuncOne(int &ref) { ... }` => 반환형이 reference인 경우에 받는 변수는 일반, ref 모두 가능!
 - ex) `int& num2 = RefRetFuncOne(num1);` (o) : 잘 사용하진 않음
 - ex) `int num2 = RefRetFuncOne(num1);` (o)
 - => return 값이 ref인 경우, 일반적으로 반환은 함수에 사용된 ref 인자여야 함
- `int RefRetFuncOne(int &ref) { ... }` => 받는 변수는 무조건 일반 value값을 가져야함
 - ex) `int& num2 = RefRetFuncOne(num1);` (x)
 - ex) `int num2 = RefRetFuncOne(num1);` (o)

* new & delete

@ new : malloc과 비슷한 의미 / delete : free와 비슷한 의미

- new : malloc을 대신하는 메모리 동적 할당 방법, 변수형으로 선언해줌 (malloc은 void 포인터)
- delete : 동적 할당된 메모리 해제

* C++에서의 구조체

@ 구조체의 등장배경

- 구조체는 연관있는 데이터를 하나로 묶는 문법적 장치
- C++에서는 구조체 변수 선언시, `struct` 키워드의 생략을 위한 `typedef` 선언이 필요하다!
- 데이터 뿐만 아니라, 해당 데이터와 연관 있는 함수들도 포함시킬 수 있음! => C++에서는 **CLASS**라 칭함

@ 구조체 안에 함수 삽입하기

- C++에서는 구조체 안에 함수 삽입이 가능 -> 함께 선언된 변수에는 직접 접근이 가능함
- 함수 본체를 선언할 수도 있지만, 프로토타입만 작성하고 외부로 뺄 수 있음 (이 때, 함수이름 앞에 소속(클래스)을 명시 해줘야함)

@ 구조체 안에 enum 상수 선언

- 구조체 안에 enum을 선언함으로써 잘못된 외부 접근을 제한할 수 있음

* 클래스(class)와 객체(object)

@ 클래스와 구조체의 차이점

- class로 선언된 멤버는 main함수에서 직접 접근이 불가능함
=> 접근제어와 관련된 함수 선언이 필요

@ 접근제어 지시자 (변수, 함수 모두 적용)

- public : 어디서든 접근 허용
- protected : 상속관계에 놓여있을 때, 유도 클래스에서의 접근 허용
- private : 클래스 내(클래스 내에 정의된 함수)에서만 접근 허용

@ 용어 정리

- 객체(object) : =클래스 / 클래스를 대상으로 생성된 변수를 객체라고 함
- 멤버 변수 : 클래스 내에 선언된 변수 (attribute)
- 멤버 함수 : 클래스 내에 정의된 함수 (method)

(8/24) DAY15 - C++

* scanf 대신 하는 데이터 입력

@ cin & cout (printf를 대신함)

- 변수 입력받을 때, 변수 타입(서식) 지정이 불필요하다는 장점
- cin은 내부적으로 reference를 사용하기 때문에 저장하는 변수에 &기호를 사용할 필요가 없는 것!

* class 사용에서 주의해야 할 것 ★★★★★

@ FS seller; => 실변수로 선언했을 경우

- buyer.BuyApples(seller, 2000); // 함수 인자에 실변수를 그대로 사용

@ FS* seller = new FS; => 포인터 타입으로 받아서 동적 할당할 경우

- buyer.BuyApples(*seller, 2000); // 실제의 값을 전달해야하기 때문에 * 키워드 사용

〈정보 은닉〉

@ 정보은닉의 이해 : 접근제어를 통해 정보의 노출을 허용하지 않음

- 멤버 변수의 외부 접근을 허용 => 정보은닉 실패 (잘못된 값이 저장 될 수 있는 문제 발생)
- 멤버 변수의 외부 접근을 막음 => 정보은닉

- ‘정보은닉’은 class의 개념이 생기면서 나온 것

=> struct(구조체)에서는 정보은닉이라는 개념이 적용되지 않음. 모든 멤버는 외부 접근이 가능함

@ 멤버함수의 const 선언 (const 함수)

- const 함수 내에서는 동일 클래스에 선언된 멤버변수 값을 변경하지 못함
- const 함수는 const가 아닌 함수를 호출하지 못함! (간접적인 멤버 변경 가능성도 완전히 차단!)
- const로 상수화 된 객체가 인자인 경우, const 멤버함수만 호출 가능

〈캡슐화 - encapsulation〉

@ 캡슐화란?

- 관련 있는 모든 것을 하나의 클래스 안에 묶어 두는 것!

〈생성자와 소멸자〉

@ 생성자

- 클래스 이름과 동일한 이름의 함수이면서 반환형이 선언되지 않았고 실제로 반환하지 않는 함수
- 생성자를 따로 정의하지 않아도, 컴파일러가 자동으로 디폴트 생성자를 추가함
- 생성자는 객체 생성시 딱 한번 호출 -> 멤버변수의 초기화에 사용할 수 있음
- 생성자도 함수의 일종 => 오버로딩이 가능하고 디폴트 값 설정 가능

@ 이니셜라이저를 이용한 변수 및 상수의 초기화

생성자 : 변수 (값)

```
class SoSimple
{
private:
    int num1;
    int num2;
public:
    SoSimple(int n1, int n2) : num1(n1)
    {
        num2 = n2;
    }
    ...
};
```

이니셜라이저를 이용한 초기화

일반 생성자를 이용한 초기화

@ 생성자 불일치

- 생성자가 삽입된 경우, 디폴트 생성자는 추가되지 않음
- => 인자를 받지 않는 void형 생성자의 호출이 불가능

@ 소멸자

- 생성자와 마찬가지로 소멸자도 정의되지 않으면 디폴트 소멸자가 삽입됨
- 생성자에서 할당한 메모리 공간을 소멸시켜야함

<클래스와 배열 그리고 this 포인터>

@ 객체 배열과 객체 포인터 배열 ★★★★★

객체 배열

```
Person arr[3];
Person* parr = new Person[3];
```

포인터 배열

```
Person *arr[3];          // Person* parr = new Person*[3];
arr[0] = new Person(name, age);
arr[1] = new Person(name, age);
arr[2] = new Person(name, age);
```

- 객체 배열 : Person(20Byte라 가정) => arr[3]은 20Byte공간 3개가 확보 = 실제 데이터가 들어가는 공간
- 포인터 배열 : 4byte(포인터 크기)공간 3개가 확보 => 주소 값이 저장됨

@ this 포인터의 이해

- 객체 자신의 포인터를 의미

〈C++에서의 static〉

@ C언어의 static => 정적 변수 / 전역 변수

- 함수 내에 선언된 static의 의미 ->> heap에 변수가 저장됨
=> 한번만 초기화되고, 지역변수와 달리 함수를 빠져나가도 소멸되지 않음

@ C++에서의 static _ class에서 사용 ->> heap에 저장됨

- static 멤버변수(클래스변수)
- static 변수는 객체별로 존재하는 변수가 아닌, 프로그램 전체 영역에서 하나만 존재하는 변수
=> 일반 멤버변수는 새롭게 선언될 때 마다 객체별로 stack의 공간을 차지하게 되지만
static 멤버변수는 heap에 하나만 존재함
- static 멤버함수 : static 멤버변수의 특징과 일치 ->> *library용 함수*
=> 생긴 것은 method 같지만, 실상은 class의 멤버함수가 아님
따라서, 멤버변수나 멤버함수에 접근이 불가능함
=> static 함수는 static 변수에만 접근 가능 / static 함수만 호출 가능

(8/25) DAY16 - C++

〈상속 - Inheritance〉

@ 상속의 방법

```
class Person :
{
private:
    int age;
    char name[50];
public:
    ...
}

class UnivStudent : public Person
{
private:
    char major[50];
public:
    ...
}
```

Person 클래스를 public 상속함

- 접근 제어 : public / private에 따라 상속 받은 하위 클래스(UnivStudent)는 접근 방법이 달라짐
- private 멤버는 유도클래스(상위, 부모)에서도 접근이 불가능
=> 생성자의 호출을 통해 기초 클래스의 멤버 초기화

@ 유도 클래스 객체의 소멸과정

- 유도 클래스의 소멸자가 실행된 이후에 기초 클래스의 소멸자가 실행
- 스택에 생성된 객체의 소멸순서는 생성순서와 반대

@ protected로 선언된 멤버가 허용하는 접근의 범위

- protected는 private과 달리 상속관계에서의 접근을 허용함!

@ 세가지 형태의 상속

- public 상속 : 접근 제어권을 그대로 상속! 단, private는 접근불가로 상속
- protected 상속 : protected보다 접근 범위가 넓은 멤버는 protected로 상속! 단, private는 접근불가로 상속
- private 상속 : private보다 접근 범위가 넓은 멤버는 private로 상속! 단, private는 접근불가로 상속

@ 상속의 기본 조건인 IS-A(~는 ~이다) 관계의 성립

- 전화기 -> 무선 전화기

=> 무선 전화기는 일종의 전화기 => 무선 전화기 is a 전화기

@ HAS-A(~는 ~을 가지고 있다) 관계 상속 => 상속 class 관계가 아님

Q. 파일을 이용해 이름, 과목명과 점수를 입력받고

과목별 합계 및 평균을 구하여 성적순으로 정렬하여 파일로 출력하시오.

단, 아래 구조를 갖는 person 클래스를 구성하고, 이를 상속하는 student 클래스를 구성하여 구현하시오.

Person)

name

age

Student) Person 상속

kor

eng

tot

avg

(8/27) DAY18 - C++

〈가상 함수 _ virtual function〉

@ 기초 클래스의 포인터로 객체 참조?

- 포인터의 자료형을 기준으로 판단 (실제 가리키는 객체의 자료형을 기준으로 판단하지 않음)

```
class Base :
{
    ...
public:
    void BaseFunc() { cout << "Base Function" << endl; }
}

class Derived : public Base
{
    ...
public:
    void DerivedFunc() { cout << "Derived Function" << endl; }
}

int main()
{
    Base* bptr = new Derived();           // 컴파일 ok
    bptr->DerivedFunc();                   // error
    Derived* dptr = bptr;                  // error

    Derived* dptr = new Derived();
    Base* bptr = dptr;                     // 컴파일 ok
}
```

〈const reference〉

@ const reference : 변경 불가

〈연산자 오버로딩〉

@ 오버로딩 불가능 연산자 종류

- 멤버 접근 연산자

(8/30) DAY19

〈review〉

@ C++ 특징

- 함수의 오버로딩 / 디폴트 / reference (* 포인터를 간편하게 사용하기 위한 것)
- 포인터와 reference의 차이
 - ~ 포인터 : 주소를 값으로 갖는 변수 (독립적인 변수)
 - ~ 레퍼런스 : 별도의 변수 자체의 주소 값을 가지지 않음. 변수의 “별명”(독립적 X)
 - 1) 함수 호출시 포인터 대신 활용
 - 2) 구조체 - class 멤버 호출 시 => 실변수 (.) / pointer(->)
- const : 변경 불가
- static 멤버 변수 : 프로그램 전체 영역에서 하나만 존재. 여러 개의 클래스에서 하나만 존재함

〈String 클래스〉

@ string 클래스 분석

- 문자열을 인자로 전달받는 생성자
- + / += 연산자(strcat()), == 연산자(strcmp()) 오버로딩으로 사용 가능
- find() : 특정 문자를 찾는 함수
- c_str() : string 타입의 문자열을 char *(char 포인터)로 반환해주는 함수

〈Template 템플릿〉

@ 함수의 오버로딩 대신 함수를 대상으로 템플릿 하자!

```
int Add (int a, int b)
{
    return a + b;
}
```

```
T Add (T a, T b)
{
    return a + b;
}
```

```
template <typename T>
T Add (T a, T b)
{
    return a + b;
}
```

```
int main()
{
    cout << Add<int>(15,20) << endl;
    cout << Add<double>(1.5,3.7) << endl;
}
```

- 기본 데이터 타입 이외의 경우? class, string도 가능하다!
- 데이터 타입 < > 은 생략 가능하지만 표준에 익숙해지기 위해 정석대로 사용하자~