안전한 Clean Code 작성을 위한 C 시큐어 코딩 실무

강사 : 정혜경, 김기희

(교재 제작 : 렉토피아)

목 차

제1장 교육과정 개설의 의미

본 과정의 목표는 C 프로그래밍 언어의 안전한 코딩을 위한 가이드 라인을 제공하여

1. 불안전한 코딩이나 취약점이 될 만한 정의되지 않은 실행을 줄이고
2. 안정성, 이식성, 신뢰성, 유지보수성, 가독성을 높여줄 수 있도록 하는것이다.

▶ 다음 코드의 결과를 예측하시요.

```c
#include <stdio.h>

int main()
{
	int x = -1;
	unsigned int y = 1;

	if (x >= y)
	{
		printf("True\n");
	}
	else
	{
		printf("False\n");
	}

	getchar();
	return 0;
}
```

세부 교육 내용

Part	교육내용
Part 1	[전처리기, 선언과 초기화 영역] 매크로 및 함수 고려사항 전역 변수 초기화와 순서 고려사항
Part 2	[표현식, 정수] 식별자 고려사항 정수 변환 규칙의 이해 정수관련 변수, 함수 및 연산자 고려사항
Part 3	[부동소수점, 배열, 문자와 문자열] 부동소수점 관련 변수, 함수 및 연산자 고려사항 배열선언 및 초기화, 사용에 대한 고려사항 문자와 문자열의 선언 및 초기화, 사요에 대한 고려사항
Part 4	[메모리 관리, 입력과 출력, 에러처리, 기타] 메모리 할당 및 해제시 고려사항 메모리 할당 에러처리 입출력 관련 함수선언 및 사용시 고려사항 에러처리시 고려사항 컴파일러 최적화

전처리기, 선언과 초기화 영역, 표현식

- ▶ 매크로 및 함수 고려사항
- ▶ 전역변수 초기화와 순서 고려사항
- ▶ 식별자 고려사항

제2장 전처리기 (PRE)

PRE-00. 함수형 매크로 보다는 인라인이나 정적 함수를 사용하라.

종류	함수 수행 원리	특징
매크로 함수	컴파일 전에 전처리기에 의해 치환	argument의 타입에 제한이 없다.
인라인 함수	함수 호출부가 함수의 몸체부로 대체됨	inline 키워드로 선언된 함수라도 inline 함수로 만들지는 컴파일러가 결정함
정적 함수	일반적인 함수로 수행됨	함수가 정의된 파일 내에서만 호출이 가능함

PRE-31. 매크로 함수 호출 시 주어지는 인자에 대한 증가, 감소, 메모리 변수 접근 등은 부수적인 효과(Side Effect)를 발생시킬 수 있다.

[문제코드]

```
#include <stdio.h>
#define CUBE(x) ((x) * (x) * (x))

int main()
{
int i = 2;
int a = CUBE(++i);

printf("a = %d\n", a);
getchar();
return 0;
}
```

[해결방법] inline 함수를 이용하여 해결

```c
#include <stdio.h>

inline int cube(int x)
{
	return x * x * x;
}

int main()
{
	int i = 2;
	int a = cube(++i);

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

[문제코드] 매크로가 사용하는 변수명과 매크로가 사용되는 블록내의 변수명이 같을 경우, 변수 사용에 문제가 발생할 수 있다.

```c
#include <stdio.h>

size_t count = 0;
#define EXEC_BUMP(func) (func(), ++count)

void g(void)
{
	printf("g() 호출, count = %u.\n", ++count);
}

void aFunc(void)
{
	size_t count = 0;
	while (count++ < 10)
	{
		EXEC_BUMP(g);
	}
}

int main()
{
	aFunc();
	getchar();
	return 0;
}
```

[해결방법] inline 함수를 이용하여 해결한다.

```c
#include <stdio.h>

size_t count = 0;

void g(void)
{
        printf("g() 호출, count = %u.\n", ++count);
}

typedef void (*exec_func)(void);
inline void exec_bump(exec_func f)
{
        f();
        ++count;
}

void aFunc(void)
{
        size_t count = 0;
        while (count++ < 10)
        {
                exec_bump(g);
        }
}

int main()
{
        aFunc();
        getchar();
        return 0;
}
```

PRE-01. 매크로 함수의 치환부에서는 매개변수에 괄호를 사용하라.

[문제코드]

```c
#include <stdio.h>
#define CUBE(I) (I * I * I)

int main()
{
        int a = CUBE(2 + 1);
        printf("a = %d\n", a);

        return 0;
}
```

[해결방법] 치환목록의 매개변수를 ()로 묶어준다.

```c
#include <stdio.h>
#define CUBE(I) ((I) * (I) * (I))

int main()
{
        int a = CUBE(2 + 1);
        printf("a = %d\n", a);

        return 0;
}
```

PRE-02. 매크로함수의 치환목록은 반드시 괄호로 둘러싸야 한다.

매크로 함수의 치환목록을 괄호로 둘러싸면 근처의 표현식으로 인해, 우선순위가 바뀌는 일을 예방할 수 있다.

[문제코드]

```c
#include <stdio.h>
#define CUBE(x) (x) * (x) * (x)

int main()
{
        int i = 3;
        int a = 81 / CUBE(i);

        printf("a = %d\n", a);
        getchar();
        return 0;
}
```

[해결방법] 치환목록을 ()로 묶어준다.

```c
#include <stdio.h>
#define CUBE(x) ((x) * (x) * (x))

int main()
{
        int i = 3;
        int a = 81 / CUBE(i);

        printf("a = %d\n", a);
        getchar();
        return 0;
}
```

[문제코드] getchar() FIN 는 getchar() -1로 치환되어 부적절하게 평가된다.

```c
#include <stdio.h>
#define FIN -1

int main()
{
        char ch;

        while ((ch = getchar()) FIN)
        {
                printf("%c", ch);
        }

        getchar();
        return 0;
}
```

[해결방법] 치환될 영역은 반드시 괄호로 둘러싸거나 열거형(enum)상수로 치환한다.

```c
#include <stdio.h>
enum { FIN = -1 };
// 또는 #define FIN (-1)

int main()
{
        char ch;

        while ((ch = getchar()) != FIN)
        {
                printf("%c", ch);
        }

        getchar();
        return 0;
}
```

PRE-03. 타입 재정의 시, 매크로 정의 대신 타입 정의를 사용하라.

자료형을 재정의 할 때에는 매크로 대신 타입정의(typedef)를 사용하라.

[문제코드] 포인터 타입을 매크로로 정의하면 부작용을 초래한다.

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

#define BUFFSIZE 100
#define cstring char *

cstring getString(const char *);

int main()
{
        cstring name, tel;

        name = getString("이름");
        tel = getString("전화");

        printf("입력된 이름 : %s\n", name);
        printf("입력된 전화번호 : %s\n", tel);

        free(name);
        free(tel);
        getchar();
        return 0;
}

cstring getString(const char *quest)
{
        cstring nr, buf = NULL;
        char tmp[BUFFSIZE];

        printf("%s : ", quest);
        if ((fgets(tmp, sizeof(tmp), stdin)) != NULL)
        {
                nr = strchr(tmp, '\n');
                *nr = nr ? '\0' : *nr;
                buf = (char *)calloc(sizeof(char), strlen(tmp) + 1);
                strncpy(buf, tmp, strlen(tmp));
        }

        return buf;
}
```

[해결방법] 타입정의(typedef)로 해결한다.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUFFSIZE 100
typedef char * cstring;

cstring getString(const char *);

int main()
{
	cstring name, tel;

	name = getString("이름");
	tel = getString("전화");

	printf("입력된 이름 : %s\n", name);
	printf("입력된 전화번호 : %s\n", tel);

	free(name);
	free(tel);
	getchar();
	return 0;
}

cstring getString(const char *quest)
{
	cstring nr, buf = NULL;
	char tmp[BUFFSIZE];

	printf("%s : ", quest);
	if ((fgets(tmp, sizeof(tmp), stdin)) != NULL)
	{
		nr = strchr(tmp, '\n');
		*nr = nr ? '\0' : *nr;
		buf = (char *)calloc(sizeof(char), strlen(tmp) + 1);
		strncpy(buf, tmp, strlen(tmp));
	}

	return buf;
}
```

PRE-05. 토큰들을 연결하거나 문자열로 변환하고자 하는 경우에는 매크로 치환을 고려하라.

- ## 연산자 (토큰병합)

  매크로가 치환되는 과정에서 두 개의 토큰을 하나로 병합(concatencation)해준다.

- # 연산자 (문자열화)

  # 연산자 다음에 있는 토큰을 문자열화 시킨다.

[예제 1] #/## 연산자

```c
#include <stdio.h>
#define MAKEVAR(x, y) x##y
#define PRN(x, y) printf(#x#y" = %d\n", MAKEVAR(x, y))

int main()
{
        int MAKEVAR(a,1) = 10;
        int MAKEVAR(a,2) = 20;

        PRN(a,1);
        PRN(a,2);

        getchar();
        return 0;
}
```

[예제 2] #/## 연산자의 활용

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

#define PRN(x, type) type##Print(x)
void charPrint(char);
void shortPrint(short);
void intPrint(int);

int main()
{
        char ch = 'A';
        short s = -1;
        int i = 2147483647;

        PRN(ch, char);
        PRN(s, short);
        PRN(i, int);

        getchar();
        return 0;
```

```
}

void charPrint(char a)
{
        unsigned char op = 1;

        op <<= sizeof(char) * CHAR_BIT - 1;
        printf("%10c : ", a);
        while (op > 0)
        {
                if (a & op) printf("1");
                else printf("0");
                op >>= 1;
        }
        printf("\n");
}

void shortPrint(short a)
{
        unsigned short op = 1;

        op <<= sizeof(short) * CHAR_BIT - 1;
        printf("%10d : ", a);
        while (op > 0)
        {
                if (a & op) printf("1");
                else printf("0");
                op >>= 1;
        }
        printf("\n");
}

void intPrint(int a)
{
        unsigned int op = 1;

        op <<= sizeof(int) * CHAR_BIT - 1;
        printf("%10d : ", a);
        while (op > 0)
        {
                if (a & op) printf("1");
                else printf("0");
                op >>= 1;
        }
        printf("\n");
}
```

[문제코드] 매크로 함수의 매개변수가 문자열화 된 후에는 치환이 불가능하다.

```
#define str(s) #s
#define foo 4

int main()
{
        printf("str(foo)의 결과 : %s\n", str(foo));
        printf("foo의 결과 : %d\n", foo);

        getchar();
        return 0;
}
```

[해결방법] 매크로 인자를 치환한 다음에 문자열로 만들려면 두 단계의 매크로를 사용해서 해결한다.

```
#include <stdio.h>
#define xstr(s) str(s)
#define str(s) #s
#define foo 4

int main()
{
        printf("xstr(foo)의 결과 : %s\n", xstr(foo));
        printf("foo의 결과 : %d\n", foo);

        getchar();
        return 0;
}
```

PRE-06. 헤더 파일에 항상 인클루전 가드를 둬라.

S/W개발 프로젝트들은 헤더파일을 포함하는 부분에서 중복하여 포함하거나 또는 미포함으로 인한 문제가 발생하기 쉽다.

macro wrapper 방식

```
#ifndef MY_H_
#define MY_H_

/* 헤더파일의 내용 */

#endif
```

#pragma directive

```
#pragma once
```

☞ macro wrapper 방식은 #ifndef 를 통해 특정 매크로가 선언되어 있는가를 확인하고, 매크로가 선언되어 있으면 #endif 까지의 코드가 무효화가 된다. 즉 일단 헤더파일을 읽어들이고 헤더 파일에 정의된 #ifndef 문의 비교가 이뤄져야 한다.

☞ #pragma once 는 전처리기에게 이 파일은 한번만 읽어들이라고 지시하는 것으로 한번 include 된 파일은 다시 include 하지 않게 된다.

☞ 그러므로 속도면에서는 #pragma once 가 macro wrapper 방식보다 더 빠른 컴파일 속도를 보인다. 하지만 #pragma once 는 표준에서 정의된 전처리 지시어가 아니므로 호환성을 위해서라면 macro wrapper 방식을 사용하는것이 좋다.

PRE-08. 중복된 헤더 파일 이름이 없는지가 보장되어야 한다.

1) 인클루드된 헤더파일의 이름이 중복되지 않도록 유의해야 한다.

- 파일의 옛 버전이 포함되어 잘못된 매크로의 정의나 예전 함수 프로토 타입으로 인한 에러가 발생할 수 있다.
- 컴파일러는 이를 발견할 수 도 있고, 찾지 못할 수 도 있다.
- 헤더 이름의 유일성이 보장되지 않는 경우, 이식성 문제까지도 유발한다.

2) 다음의 규칙을 고려하여 헤더 파일명을 지정하여야 한다.

- 처음 여덟 글자만 헤더 파일 이름으로 보장할 수 있다.
- 파일명의 점 뒤에는 숫자가 아닌 문자 한 개만 올 수 있다.
- 파일명에서 대소문자 구별은 보장하지 않는다.

PRE-10. 복수 구문의 매크로는 do ~ while 루프로 감싼다.

[문제코드]

```c
#include <stdio.h>
#define SWAP(x,y) \
        tmp = x; \
        x = y; \
        y = tmp

int main()
{
        int x, y, z, tmp = 0;

        printf("x = "); scanf("%d", &x);
        printf("y = "); scanf("%d", &y);
        printf("z = "); scanf("%d", &z);

        if (z == 0) SWAP(x, y);
        printf("x = %d, y = %d, z = %d\n", x, y, z);
        return 0;
}
```

[해결방법]

```c
#include <stdio.h>
#define SWAP(x,y) \
do { \
        tmp = x; \
        x = y; \
        y = tmp; \
} while (0)

int main()
{
        int x, y, z, tmp = 0;

        printf("x = "); scanf("%d", &x);
        printf("y = "); scanf("%d", &y);
        printf("z = "); scanf("%d", &z);

        if (z == 0) SWAP(x, y);
        printf("x = %d, y = %d, z = %d\n", x, y, z);
        return 0;
}
```

제3장 선언과 초기화

DCL-00. 변하지 않는 값이 저장된 변수는 const 로 보장해줘라.

변수의 불변성을 const로 보장하면 프로그램의 정확성과 안전성을 보장하는데 도움이 된다. 변수를 const로 선언하는 대신 매크로나 열거형 상수를 사용할 수 있다.

[문제코드] pi는 수학적으로 원주율을 의미하는 불변의 상수인데 코드에서는 이 값이 변하지 못하도록 보호되고 있지 않다.

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

int main()
{
        double pi = 3.14159;
        double radius;
        double area;

        printf("반지름 : ");
        if (scanf("%lf", &radius) != 1)
        {
                printf("입력오류!!!\n");
                return 1;
        }

        area = pi * radius * radius;
        printf("원의 면적 : %lf\n", area);

        __fpurge(stdin);
        getchar();
        return 0;
}
```

[해결방법] pi를 const로 선언하여 해결한다.

```c
#include <stdio.h>
#include <stdio_ext.h>
/*
#define PI 3.14159
*/

int main()
{
        const double pi = 3.14159;
        double radius;
        double area;
```

```
        printf("반지름 : ");
        if (scanf("%lf", &radius) != 1)
        {
                printf("입력오류!!!\n");
                return 1;
        }

        area = pi * radius * radius;
        printf("원의 면적 : %lf\n", area);

        __fpurge(stdin);
        getchar();
        return 0;
}
```

DCL-01. 내부영역에서 변수의 이름을 재사용하지 마라.

ㅇ 한 영역이 다른 영역 안에 포함되어 있는 경우, 두 영역에서 동일한 변수의 이름을 사용하지 마라.

ㅇ extern변수가 사용될 수 있는 범위 내에서 extern변수명과 동일한 auto변수명을 사용하면 안된다.

ㅇ 중첩되지 않는 블록 사이에서도 동일한 이름의 변수명을 사용하지 않도록 하는것이 좋다.

ㅇ 변수명이 재사용이 될 경우, 프로그래머의 입장에서는 어떤 변수가 변경되고 있는지 혼동할 수 가 있다.

[문제코드] 파일 전체에서 사용가능한 extern msg배열과 같은 이름의 배열을 report_error()함수에서 선언하여 사용하므로써 extern msg배열의 초기화에 실패할 뿐만 아니라 잠정적으로 버퍼 오버플로우의 발생도 가능하다.

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

char msg[100];
void report_error(const char *);

int main()
{
        const double pi = 3.14159;
        double radius;
        double area;

        printf("반지름 : ");
```

```
        if (scanf("%lf", &radius) != 1)
        {
                report_error("데이터 입력오류");
                return 1;
        }

        area = pi * radius * radius;
        printf("원의 면적 : %lf\n", area);

        __fpurge(stdin);
        getchar();
        return 0;
}

void report_error(const char *error_msg)
{
        char msg[80];

        memset(msg, '\0', sizeof(msg));
        strncpy(msg, error_msg, strlen(error_msg));
}
```

[해결방법] extern변수명과 auto변수명을 다른 용도로 사용하고, 변수명도 다르게 사용하도록 한다.

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

char msg[100];
void report_error(const char *);

int main()
{
        const double pi = 3.14159;
        double radius;
        double area;

        printf("반지름 : ");
        if (scanf("%lf", &radius) != 1)
        {
                report_error("입력오류가 발생하였습니다.");
                printf("msg : %s\n", msg);
                return 1;
        }

        area = pi * radius * radius;
        printf("원의 면적 : %lf\n", area);

        __fpurge(stdin);
        getchar();
        return 0;
```

```
}

void report_error(const char *error_msg)
{
        char default_msg[] = "알 수 없는 오류가 발생하였습니다.";

        memset(msg, '\0', sizeof(msg));
        if (error_msg == NULL || strlen(error_msg))
        {
                strncpy(msg, error_msg, strlen(error_msg));
        }
        else
        {
                strncpy(msg, default_msg, strlen(default_msg));
        }
}
```

DCL-02. 시각적으로 구별되는 식별자를 사용하라.

ㅇ 여러 식별자를 끝의 한 두 글자만 다르게 해서 구별할 때, 시각적으로 비슷해 보이는 문자를 구별되는 문자로 사용하지 마라.

- 1(하나)과 l(소문자 L)
- 0(숫자 영)가 O(대문자 O)
- 2(둘)과 Z(대문자 Z)
- 5(다섯)과 S(대문자 S)
- 8(여덟)과 B(대문자 B)

[문제코드] 아래의 두 변수는 혼동되기 쉽다.

```
        int number1;
        int numberl;
```

DCL-05. 코드의 가독성을 높이기 위해 타입 정의를 사용하라.

☞ signal()함수는 아래와 같이 선언되어 있다.

```
void (*signal(int signum, void (*handler)(int)))(int);
```

이것은 signal()함수는 아래와 같이 두개의 파라미터를 가지며

```
signal(int signum, void (*handler))
```

signal()함수가 반환하는 값은 아래와 같은 함수포인터형임을 나타낸다.

```
void (*)(int);
```

위의 signal()함수는 아래와 같이 typedef를 이용하여 함수 포인터 타입을 정의하여 사용하면 코드의 가독성을 높일 수 있다.

```
typedef void (*sigHandlerType)(int);
extern sigHandlerType signal(int signum, sigHandlerType handler);
```

[문제코드] 아래의 예제에서 `getFunction()` 함수에 대한 선언은 읽기도 어렵고 이해하기도 힘들다.

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

int (*getFunction(int))(void);
int getFruits();
int getGrains();

int main()
{
        int (*func)(void);
        func = getFunction(1);
        func();
        func = getFunction(2);
        func();

        __fpurge(stdin);
        getchar();
        return 0;
}
```

```
int (*getFunction(int type))(void)
{
        int (*func)(void);

        switch (type)
        {
        case 1:
                func = getFruits;
                break;
        case 2:
                func = getGrains;
                break;
        }

        return func;
}

int getFruits()
{
        printf("getFruits() 함수가 선택되었습니다.\n");
        return 1;
}

int getGrains()
{
        printf("getGrains() 함수가 선택되었습니다.\n");
        return 1;
}
```

[해결방법] typedef 를 이용하여 함수 포인터를 정의한 후, 사용

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

typedef int (*Func)(void);
Func getFunction(int);
int getFruits();
int getGrains();

int main()
{
        Func func;
        func = getFunction(1);
        func();
        func = getFunction(2);
        func();

        __fpurge(stdin);
        getchar();
        return 0;
}
```

```
Func getFunction(int type)
{
	Func func;

	switch (type)
	{
	case 1:
		func = getFruits;
		break;
	case 2:
		func = getGrains;
		break;
	}

	return func;
}

int getFruits()
{
	printf("getFruits() 함수가 선택되었습니다.\n");
	return 1;
}

int getGrains()
{
	printf("getGrains() 함수가 선택되었습니다.\n");
	return 1;
}
```

DCL-06. 프로그램 로직상의 고정적인 값을 나타낼 때에는 의미있는 심볼릭 상수를 사용하라.

ㅇ C언어에서는 리터럴 상수와 심볼릭 상수를 제공한다.

ㅇ 리터럴 상수는 소스코드의 가독성을 떨어뜨리며, 값을 변경할 경우, 비 효율적으로 수정되는 현상이 나타므로 리터럴 상수를 직접 표기(하드코딩)하기 보다는 심볼릭 상수를 사용하는것이 가독성을 높이고, 유지보수를 쉽게 할 수 있다.

ㅇ C언어에서의 3가지 심볼릭 상수

- 매크로 상수
- 열거형 상수
- const 상수

☞ const 상수

ㅇ const 상수는 const 상수가 선언된 블럭 내에서만 사용이 가능하다.

ㅇ 컴파일러에 의해 type이 체크된다.

ㅇ 디버깅 도구로 디버깅이 가능하다.

ㅇ 함수 내부에서 사용하면 함수 호출시 마다 할당되고 초기화 되는 오버 헤드가 발생한다.

const 상수는 컴파일 타임에서 정수형 상수가 필요한 곳에서는 사용할 수 없다.

[문제코드] const 상수는 구조체 맴버의 비트 크기로 사용할 수 없다.

```
#include <stdio.h>

const int bit_size = 3;

typedef struct _bitfield
{
        int a : 3;
        unsigned int b : bit_size;
} BitField;

int main()
{
        BitField bf = {-3, 7};

        printf("bf.a : %d\n", bf.a);
        printf("bf.b : %u\n", bf.b);

        getchar();
```

```
        return 0;
}
```

[해결방법] 매크로 상수나 열거형 상수를 사용

```
#include <stdio.h>

// #define bit_size 3
enum { bit_size=3 };

typedef struct _bitfield
{
        int a : 3;
        unsigned int b : bit_size;
} BitField;

int main()
{
        BitField bf = {-3, 7};

        printf("bf.a : %d\n", bf.a);
        printf("bf.b : %u\n", bf.b);

        getchar();
        return 0;
}
```

[문제코드] const 상수는 배열원소의 갯수로 사용할 수 없다.

```
#include <stdio.h>

const int array_size = 3;

int main()
{
        int i;
        int arr[array_size] = {1, 2, 3};
        size_t size = sizeof(arr)/sizeof(arr[0]);

        for (i=0 ; i<size ; i++)
        {
                printf("arr[%d] : %d\n", i, arr[i]);
        }

        getchar();
        return 0;
```

```
}
```

[해결방법] 매크로 상수나 열거형 상수를 사용

```
#include <stdio.h>

//#define array_size 3
enum { array_size=3 };

int main()
{
        int i;
        int arr[array_size] = {1, 2, 3};
        size_t size = sizeof(arr)/sizeof(arr[0]);

        for (i=0 ; i<size ; i++)
        {
                printf("arr[%d] : %d\n", i, arr[i]);
        }

        getchar();
        return 0;
}
```

[문제코드] const 상수는 switch문의 case 값으로 사용할 수 없다.

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

const int strawberry = 3;
enum FRUITS { APPLE, CHERRY, ORANGE };

int main()
{
        int item;
        printf("상품번호를 입력하시오 (0 ~ 3) : ");
        if (scanf("%d", &item) != 1){
                printf("입력오류!!!\n");
                return 1;
        }

        switch (item)
        {
                case APPLE:
                        fputs("APPLE choice!", stdout); break;
                case CHERRY:
                        fputs("CHERRY choice!", stdout); break;
```

```
                case ORANGE:
                        fputs("ORANGE choice!", stdout); break;
                case strawberry:
                        fputs("STRAWBERRY choice!", stdout); break;
        }

        __fpurge(stdin);
        getchar();
        return 0;
}
```

[해결방법]

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

enum FRUITS { APPLE, CHERRY, ORANGE, strawberry };

int main()
{
        int item;
        printf("상품번호를 입력하시오 (0 ~ 3) : ");
        if (scanf("%d", &item) != 1){
                printf("입력오류!!!\n");
                return 1;
        }

        switch (item)
        {
                case APPLE:
                        fputs("APPLE choice!", stdout);
                        break;
                case CHERRY:
                        fputs("CHERRY choice!", stdout);
                        break;
                case ORANGE:
                        fputs("ORANGE choice!", stdout);
                        break;
                case strawberry:
                        fputs("STRAWBERRY choice!", stdout);
                        break;
        }

        __fpurge(stdin);
        getchar();
        return 0;
}
```

☞ 열거형 상수

- ㅇ 열거형 상수는 int로 나타낼 수 있는 정수형 상수 표현식을 나타낼 때 사용한다.
- ㅇ 별도의 메모리가 할당되지 않는다. (열거형 상수의 주소를 구할 수 없음)
- ㅇ 열거형 상수는 type을 따로 지정할 수 없다. (항상 int형이다.)

```c
#include <stdio.h>

enum { MAX=15 };

int main()
{
        int ary[MAX];    // 열거형 상수로 배열의 크기 지정 가능
        const int *p;
        p = ary;
        p = &MAX;        // 열거형 상수는 주소 계산이 불가능

        return 0;
}
```

☞ 객체형 매크로

```
#define identifier replacement-list
```

객체형 매크로 (Object-like Macro)는 전처리기(cpp)에 의해 프로그램 소스상의 매크로 이름(identifier)이 지시문의 나머지 부분(replacement-list)로 치환된다.

- ㅇ 객체형 매크로는 메모리를 소비하지 않으므로 포인터로 가리킬 수 없다.
- ㅇ 전처리기는 컴파일러가 심볼을 처리하기 전에 매크로 치환작업을 한다.
- ㅇ 매크로는 타입 체크도 제공하지 않는다.

DCL-06.1. 프로그램 로직상의 고정적인 값을 타나낼 때에는 의미 있는 심볼릭 상수를 사용하라.

[문제코드] 아래 코드에 있는 정수형 리터럴 18의 의미가 분명치 않다.

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

int main()
{
        int age;

        printf("당신의 나이를 입력하시오 : ");
        if (scanf("%d", &age) != 1)
        {
                printf("데이터입력오류!\n");
                return 1;
        }

        if (age >= 18)
        {
                printf("당신은 성인입니다.\n");
        }
        else
        {
                printf("당신은 미성년입니다.\n");
        }

        __fpurge(stdin);
        getchar();
        return 0;
}
```

[해결방법] 정수형 리터럴 18을 심볼릭 상수로 선언(의미를 분명하게)하여 사용한다.

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

enum { ADULT_AGE=18 };

int main()
{
        int age;

        printf("당신의 나이를 입력하시오 : ");
        if (scanf("%d", &age) != 1)
        {
```

```
                printf("데이터입력오류!\n");
                return 1;
        }

        if (age >= ADULT_AGE)
        {
                printf("당신은 성인입니다.\n");
        }
        else
        {
                printf("당신은 미성년입니다.\n");
        }

        __fpurge(stdin);
        getchar();
        return 0;
}
```

[문제코드] buffer의 크기가 바뀌면 fgets()함수 호출부분도 바뀌어야 한다.

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

int main()
{
        char *nr;
        char buffer[256];

        printf("이름 : ");
        if (fgets(buffer, 256, stdin) == NULL)
        {
                printf("입력오류발생!\n");
                return 1;
        }

        nr = strchr(buffer, '\n');
        *nr = nr ? '\0' : *nr;
        printf("입력된 이름 : %s\n", buffer);

        __fpurge(stdin);
        getchar();
        return 0;
}
```

[해결방법 1] 매크로 상수나 열거형 상수를 사용

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

enum { BUFFER_SIZE=256 };

int main()
{
        char *nr;
        char buffer[BUFFER_SIZE];

        printf("이름 : ");
        if (fgets(buffer, BUFFER_SIZE, stdin) == NULL)
        {
                printf("입력오류발생!\n");
                return 1;
        }

        nr = strchr(buffer, '\n');
        *nr = nr ? '\0' : *nr;
        printf("입력된 이름 : %s\n", buffer);

        __fpurge(stdin);
        getchar();
        return 0;
}
```

[해결방법 2] sizeof 연산자를 사용 (※ 아무런 문제가 없는가? buffer가 동적할당이라면?)

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

int main()
{
        char *nr;
        char buffer[256];

        printf("이름 : ");
        if (fgets(buffer, sizeof(buffer), stdin) == NULL)
        {
                printf("입력오류발생!\n");
                return 1;
        }

        nr = strchr(buffer, '\n');
        *nr = nr ? '\0' : *nr;
```

```
        printf("입력된 이름 : %s\n", buffer);

        __fpurge(stdin);
        getchar();
        return 0;
}
```

DCL-07. 함수 선언 시 적절한 타입 정보를 포함시켜라.

DCL-35. 함수 정의와 맞지 않는 타입으로 함수를 변환하지 마라.

아래의 예제에서 함수포인터 func의 타입은 add()함수의 타입과 일치하지 않는다.
즉 함수포인터 func타임은 int형 파라미터가 두개이므로 호출시 2개의 인자만 넘길 수 있으며, func가 가리키는 실제 함수인 add는 파라미터가 3개이므로 마지막 파라미터의 값은 쓰레기 값이 들어있거나 0일 수도 있다.

[문제코드]

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

int add(int, int, int);

int main()
{
        int res;
        int (*func)(int, int);

        func = add;
        res = func(10, 20);
        printf("res : %d\n", res);

        __fpurge(stdin);
        getchar();
        return 0;
}

int add(int x, int y, int z)
{
        return x + y + z;
}
```

[해결방법] 함수 프로토타입과 함수 포인터 타입을 일치시킨다.

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

int add(int, int, int);

int main()
{
        int res;
        int (*func)(int, int, int);

        func = add;
        res = func(10, 20, 30);
        printf("res : %d\n", res);

        __fpurge(stdin);
        getchar();
        return 0;
}

int add(int x, int y, int z)
{
        return x + y + z;
}
```

DCL-08. 상수 정의에서는 상수 간의 관계가 적절하게 나타나도록 정의하라.

한 정의가 다른 정의에 영향을 미친다면 둘 간의 관계를 인코딩(부호화)하고 각각을 따로 정의하지 마라. 또한 서로 관계가 없는 것들을 인코딩 하지마라.

[문제코드] OUT_STR_LEN은 항상 IN_STR_LEN보다 커야 한다면

```c
enum { IN_STR_LEN=18, OUT_STR_LEN=20 };
```

[해결방법] 두 정의간의 관계를 명시적으로 나타내어 표현한다.

```c
enum { IN_STR_LEN=18, OUT_STR_LEN=IN_STR_LEN+2 };
```

[문제코드] 서로에게 상관없는 관계가 성립되어 있다.

```
enum { ADULT_AGE = 18 };
enum { ALCOHOL_AGE = ADULT_AGE + 3 };
```

[해결방법] 상관없는 관계는 정의에 포함하지 않는다.

```
enum { ADULT_AGE = 18 };
enum { ALCOHOL_AGE = 21 };
```

DCL-10. 가변인자 함수를 사용시에는 함수 작성자와 함수 사용가간의 약속이 지켜져야 한다.

가변인자함수란? 인수의 개수가 정해지지 않은 함수를 말한다. 예를들어 printf()함수나 scanf() 함수등은 출력 포맷을 고정인수로 하며, 고정인수 뒤에는 출력할 변수나, 데이터를 가리키는 포인터 또는 데이터를 저장할 포인터 등이 나열되게 된다.

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

이와같이 함수 형식인수의 개수가 정해져 있지 않은 함수를 가변인수 함수라고 한다.
이러한 가변인수 함수의 사용은 함수 작성자와 사용자간에 규칙을 정하고 이 규칙에 의해 작성하고 사용하여야만 한다.

가변인수 함수의 특징은 반드시 하나 이상의 고정인수를 가져야만 한다는 것이다. 이것은 함수 호출시 전달된 인자가 스택에 저장되고, 이렇게 스택상에 저장된 인수의 위치를 가리키기 위함이다.

[예제] 가변인수 함수 예제 (아래 예제의 실행결과는?)

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

enum {va_eol = -1};
double average(int first,...);

int main()
{
    double res;

res = average(10,30,va_eol);
printf("res = %.2lf\n", res);

res = average(-7,-5,-1,1,5,7,va_eol);
printf("res = %.2lf\n", res);


        getchar();
        return 0;
}

double average(int first,...)
{
        unsigned int count = 0;
        unsigned int sum = 0;
        int  i = first;

        va_list  args;
        va_start(args, first);
        while(i!=va_eol)
        {
                sum += i;
                count++;
                i = va_arg(args, int);
        }
        va_end(args);

        return (count ? ( (double) sum/count) : 0);
}
```

1. va_list args;

함수로 전달되어 오는 전달인자들은 스택에 저장되며, 함수는 스택에서 인수를 꺼내 사용하게 된다. 이때 스택에 저장되어 있는 인수를 args를 이용하여 읽어내게 된다.

va_list : char *형

args : argument를 가리키는 포인터

2. va_start(args, 마지막고정인수)

가변인수를 읽기 위한 준비 함수로써 args 포인터가 첫번째 가변인수를 가리키도록 초기화 한다.

3. va_arg(args, type)

실제 가변인수의 값을 읽어들이는 함수로써 args가 가리키는 위치의 값을 두번째 파라미터로 지정된 type형으로 읽어들인다.

4. va_arg(args)

매크로 함수는 뒷정리용 함수로써 intel계열의 시스템에서는 아무런 작동도 하지 않는다. 하지만 이 명령이 필요한 이유는 호환성 때문인데 플랫폼에 따라 가변인수를 읽은 후 뒷처리를 해주어야 할 필요가 있기 때문이다. 정확하게는 가변인수를 읽은 후 뒷처리를 하는 플랫폼이 있다 것이다. 그러므로 관례적으로 호출하는것이 좋다.

DCL-15. 블럭 범위를 넘어서까지 사용되지 않을 객체는 static으로 선언하라.

유형	설명
변수	선언된 블럭에서만 접근이 가능하다.
함수	정의된 파일 내에서만 접근이 가능하다.

[문제코드] helper() 함수는 오직 util.c에서만 호출이 가능하여야만 한다.

```
[util.h 의 내용]
#ifndef UTIL_H_
#define UTIL_H_

int helper(int);

#endif

[util.c 의 내용]
int helper(int i)
{
        printf("이 함수는 오직 해당 파일에서만 호출이 가능하여야 합니다.\n");
}

void func()
{
        helper(3);
}

[main.c 의 내용]
#include <stdio.h>
#include "util.h"
```

```
int main()
{
        printf("helper()를 호출합니다.\n");
        helper(1);
        printf("func()를 호출합니다.\n");
        func();

        getchar();
        return 0;
}
```

[해결방법] helper() 함수를 static 키워드를 이용하여 정의함.

[util.h 의 내용]

```
#ifndef UTIL_H_
#define UTIL_H_

int helper(int);

#endif
```

[util.c 의 내용]

```
static int helper(int i)
{
        printf("이 함수는 오직 해당 파일에서만 호출이 가능하여야 합니다.\n");
}

void func()
{
        helper(3);
}
```

[main.c 의 내용]

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

int main()
{
        printf("helper()를 호출합니다.\n");
        helper(1);
        printf("func()를 호출합니다.\n");
        func();

        getchar();
        return 0;
}
```

[예제] 생각해볼 문제 (static 변수는 외부에서 접근이 불가능 한가?)

```c
#include <stdio.h>
#include <string.h>
int ary[5] = {10, 20, 30, 40, 50};

void sub()
{
        static int snum = 3;
        printf("snum의 값\t: %d\n", snum++);
        printf("snum의 주소\t: %p\n", &snum);
}

int main()
{
        ary[5]++;
        sub();
        ary[5]++;
        sub();

        getchar();
        return 0;
}
```

DCL-30. 적절한 기억클래스를 사용하여라.

○ 변수는 프로그램 수행 시, 자신에게 할당된 저장공간에 존재하다가 마지막으로 저장되어 있던 값을 유지한 채 수명이 다하게 된다.

○ 그러므로 변수가 수명이 다한 후에도 참조된다면, 정의되지 않은 행동을 유발할 수 있다.

○ 수명을 다한 데이터 가리키는 포인터로 데이터에 접근하는 일은 악용될 수 있는 취약성을 만드는 결과를 초래하기도 한다.

[문제코드]

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

void dont_do_this();
void innocuous();

const char *p;

int main()
{
        dont_do_this();
        innocuous();
        printf("p=%s\n", p);

        getchar();
        return 0;
}

void dont_do_this()
{
        const char str[] = "This will change";
        p = str;
        printf("p=%s : str=%s\n", p, str);
}

void innocuous()
{
        const char str[]="Surprise, surprise";
        printf("p=%s : str=%s\n", p, str);
        /* p에 의해 출력되는 값은? */
}
```

[해결방법 1]

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

void dont_do_this();
void innocuous();

const char *p;

int main()
{
        dont_do_this();
        innocuous();
        printf("p=%s\n", p);

        getchar();
        return 0;
}

void dont_do_this()
{
        const char str[] = "This will change";
        p = str;
        printf("p=%s : str=%s\n", p, str);
        p = NULL;
}

void innocuous()
{
        const char str[]="Surprise, surprise";
        printf("p=%s : str=%s\n", p, str);
        /* p에 의해 출력되는 값은? */
}
```

[해결방법 2]

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

void dont_do_this();
void innocuous();

int main()
{
        dont_do_this();
        innocuous();
        // p는 auto변수이므로 참조가 불가능
        // printf("p=%s\n", p);
```

```
        getchar();
        return 0;
}

void dont_do_this()
{
        const char str[] = "This will change";
        const char *p;
        p = str;
        printf("p=%s : str=%s\n", p, str);
        p = NULL;
}

void innocuous()
{
        const char str[]="Surprise, surprise";
        printf("str=%s\n", str);
}
```

DCL-33. 함수 인자에서 restrict로 지정된 소스 포인터와 목적 포인터가 동일한 객체를 참조하지 않게 하라.

restrict 지정자로 지정된 함수의 두 인자가 동일한 공간을 참조하는 경우에는 그 결과를 예측할 수 없다.

ㅇ restrict 키워드

포인터에만 사용가능한 지정자로써 컴파일러가 특정 유형의 코드를 최적화 할 수 있도록 허용함으로써 연산능력을 향상시킨다.

restrict로 지정된 포인터는 그 포인터가 포인터가 가리키는 공간에 접근할 수 있는 유일한 최초의 수단이라는 것을 나타낸다.

몇몇 C99 함수는 restrict 지정자를 사용한 매개변수를 갖는 함수들이 있다.

☞ 아래의 예제는 일부 컴파일러에서 문제를 발생시킬 수 있다.

[문제코드] ptr1과 ptr2는 같은 배열을 가리킨다.

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

int main()
{
        char str[] = "test string";
        char *ptr1 = str;
        char *ptr2 = ptr1 + 3;

        memcpy(ptr2, ptr1, 6);
        printf("ptr1 : %s\n", ptr1);
        printf("ptr2 : %s\n", ptr2);

        getchar();
        return 0;
}
```

[해결방법] memcpy() 함수 대신 memmove() 함수를 사용

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

int main()
{
        char str[] = "test string";
        char *ptr1 = str;
        char *ptr2 = ptr1 + 3;

        memmove(ptr2, ptr1, 6);
        printf("ptr1 : %s\n", ptr1);
        printf("ptr2 : %s\n", ptr2);

        getchar();
        return 0;
}
```

DCL-34. cache되어서는 안되는 데이터에는 volatile을 사용하라.

volatile은 어떤 변수가 해당 프로그램이 아닌 다른 대행자에 의해 그 값이 변경될 수 있다고 컴파일러에게 알린다.

volatile로 지정된 변수는 하드웨어 주소 또는 동시에 실행되는 여러 프로그램들이 공유하는 데이터에 사용된다.

```
num1 = x;
/* x를 사용하지 않는 코드 부분 */
......
num2 = y;
```

위의 코드에서 컴파일러는 x의 값을 변경하지 않고 두 번 사용하고 있으므로 x의 값을 레지스터에 임시로 저장해 놓고 val2에 x를 대입할 때 원래의 x의 값 대신에 레지스터에서 그 값을 읽음으로써 시간을 절약할 수 있도록 최적화를 하게 된다. 이것을 캐싱(caching)이라고 함.

```
num += 1;
/* num을 사용하지 않는 코드부분 */
num += 2;
/* num을 사용하지 않는 코드부분 */
num += 3;
```

위의 코드에서 변수 num에 값을 누적시키고 있으나 num의 값을 사용하는 코드 부분이 없으므로 아래와 같이 최적화를 시킨다.

```
/* num += 1; */
/* num을 사용하지 않는 코드부분 */
/* num += 2; */
/* num을 사용하지 않는 코드부분 */
num += 6;
```

제4장 표현식(EXP)

EXP-00. 연산자 우선순위를 나타내는 데 괄호를 사용하라.

[부적절한 코드의 예]

```
x & 1 == 0              ◀ x의 최하위 비트를 테스트 하려하는가?

int *p;
int num = 3;
p = &num;
*p++;                   ◀ p가 가리키는것을 증가하려 하는가?

int num = 10;
res = !num == 10;       ◀ num과 10을 비교 후, 논리 부정 하려 하는가?
```

[부적절한 코드의 예]

```
(x & 1) == 0            ◀ x의 최하위 비트가 0인가를 테스트 한다.

int *p;
int num = 3;
p = &num;
(*p)++;                 ◀ p가 가리키는 곳의 값을 사용한 후, 증가시킨다.

int num = 10;
res = !(num == 10);     ◀ num이 10과 같지 않은가를 테스트 한다.
```

▶ 연산자 우선 순위표

연산자 우선순위와 동일우선 순위에 적용되는 결합방향은 수식의 연산순서를 결정하는 중요한 문법이다.

순위	명칭		연산자	결합방향
1	1차 연산자		() [] . ->	->
2	단항 연산자		+ - ! ~ (type) sizeof ++ -- & *	<-
3	이항연산자	승법 연산자	* / %	->
4		가법연산자	+ -	
5		Shift 연산자	<< >>	
6		관계 연산자	< > <= >=	
7		등가 연산자	== !=	
8		bit 곱 연산자	&	
9		bit 차 연산자	^	
10		bit 합 연산자	|	
11		논리곱 연산자	&&	
12		논리합 연산자	||	
13	조건 연산자		?:	->
14	대입 연산자		= += -= *= /= %= <<= >>= &= |=	<-
15	순차 연산자		,	->

[예제] 다음 코드의 실행결과를 기술하시오.

```c
#include <stdio.h>

int main()
{
	int *p;
	int num[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
	p = num;

	printf("p : %p\n", p);
	printf("*p : %d\n", *p);

	*p++;
	printf("p : %p\n", p);
	printf("*p : %d\n", *p);

	(*p)++;
	printf("p : %p\n", p);
	printf("*p : %d\n", *p);

	p = num + sizeof(num)/sizeof(num[0])/2;
	printf("p : %p\n", p);
	printf("*p : %d\n", *p);

	*++p;
	printf("p : %p\n", p);
	printf("*p : %d\n", *p);

	++*p;
	printf("p : %p\n", p);
	printf("*p : %d\n", *p);

	getchar();
	return 0;
}
```

EXP-02. 논리 연산자 AND와 OR의 단축 평가 방식을 알고 있어라.

논리 AND와 논리 OR 연산자는 첫 번째 피연산자로 평가가 완료되면 두번째 피 연산자는 평가하지 않는다.

연산자	좌측피연산자	우측피연산자
AND	거짓 인 경우	평가 안함
OR	참 인 경우	평가 안함

[문제코드]

```
#include <stdio.h>
#include <stdio_ext.h>
#include <string.h>
#include <malloc.h>
#define BUF_SIZE 20

int getString(const char *, char **);

int main()
{
        int res;
        char *p;

        /* p는 NULL일수도 있고 아닐 수도 있다. */
        if (p || (p = (char *)malloc(BUF_SIZE)))
        {
                res = getString("이름입력", &p);
                if (res == 1)
                {
                        printf("입력된 이름은 '%s'입니다.", p);
                }
                free(p);
                p = NULL;
        }
        else
        {
                printf("동적메모리 할당에러!!!\n");
                return 1;
        }

        __fpurge(stdin);
        getchar();
        return 0;
}


int getString(const char *q, char **p)
{
        char *nr = NULL;
```

```
        printf("%s : ", q);
        if (fgets(*p, BUF_SIZE, stdin) == NULL)
        {
                return 0;
        }
        nr = strchr(*p, '\n');
        if (nr != NULL) *nr = '\0';

        return 1;
}
```

[해결방법]

```
#include <stdio.h>
#include <stdio_ext.h>
#include <string.h>
#include <malloc.h>
#define BUF_SIZE 20

int getString(const char *, char **);

int main()
{
        int res;
        char *p;

        /* p가 NULL이 아니라면 초기화 하라. */

        p = (char *)malloc(BUF_SIZE);
        if (p == NULL)
        {
                printf("동적메모리 할당에러!!!\n");
                return 1;
        }

        res = getString("이름입력", &p);
        if (res == 1)
        {
                printf("입력된 이름은 '%s'입니다.", p);
        }
        free(p);
        p = NULL;

        __fpurge(stdin);
        getchar();
        return 0;
}

int getString(const char *q, char **p)
{
        char *nr = NULL;
```

```
        printf("%s : ", q);
        if (fgets(*p, BUF_SIZE, stdin) == NULL)
        {
                return 0;
        }
        nr = strchr(*p, '\n');
        if (nr != NULL) *nr = '\0';

        return 1;
}
```

EXP-03. 구조체의 크기가 구조체 맴버들 크기의 합이라고 가정하지 마라.

EXP-04. 구조체끼리 바이트 단위로 비교하지 마라.

구조체 변수를 선언하여 메모리에 할당될 때에는 구조체 정렬이 발생한다.

```
struct _person
{
        char name[24];
        char flag
        int age;
        unsigned long id;
}
```

위의 구조체는 구조체 정렬에 의해 32비트 시스템에서는 36바이트를 할당 받지만 64비트 시스템에서는 40바이트의 메모리 공간을 할당받게 된다.

32bit 시스템

name[24]	flag	패딩	age	id
24 Byte	1Byte	3Byte	4Byte	4Byte

64bit 시스템

name[24]	flag	패딩	age	id
24 Byte	1Byte	3Byte	4Byte	8Byte

구조체 정렬은 구조체 맴버중 가장 큰 자료형을 기준으로 정렬이 이뤄진다. 만약 맴버중 가장

큰 자료형의 크기가 4Byte미만인 경우에는 4Byte단위로 정렬을 하게 된다. 그러므로 구조체의 크기는 시스템에 따라 할당받는 메모리 공간의 크기가 다르며, 맴버의 순서에 의해서도 크기가 달라지게 된다.

구조체의 크기는 sizeof(구조체)이다.

[문제코드]

```
#include <stdio.h>
#include <stdio_ext.h>
#include <malloc.h>
#include <string.h>

typedef struct _Person
{
        char name[30];
        int age;
} Person;

int main()
{
        Person *p1;

        p1 = (Person *)malloc(34);
        if (p1 == NULL)
        {
                printf("동적메모리 할당 실패!!!\n");
                return 1;
        }


        strcpy(p1->name, "KIHEE KIM");
        p1->age = 20;

        printf("NAME : %s\n", p1->name);
        printf("AGE : %d\n", p1->age);

        free(p1);
        p1 = NULL;

        getchar();
        return 0;
}
```

[해결방법]

```c
#include <stdio.h>
#include <stdio_ext.h>
#include <malloc.h>
#include <string.h>

typedef struct _Person
{
        char name[30];
        int age;
} Person;

int main()
{
        Person *p1;

        p1 = (Person *)malloc(sizeof(Person));
        if (p1 == NULL)
        {
                printf("동적메모리 할당 실패!!!\n");
                return 1;
        }


        strcpy(p1->name, "KIHEE KIM");
        p1->age = 20;

        printf("NAME : %s\n", p1->name);
        printf("AGE : %d\n", p1->age);

        free(p1);
        p1 = NULL;

        getchar();
        return 0;
}
```

EXP-06. sizeof의 피연산자가 다른 부수 효과를 가지면 안된다.

[문제코드]

```c
#include <stdio.h>

int main()
{
	int a;
	int b;

	a = 14;
	b = sizeof(a++);

	printf("a : %d\n", a);
	printf("b : %d\n", b);

	getchar();
	return 0;
}
```

[해결방법]

```c
#include <stdio.h>

int main()
{
	int a;
	int b;

	a = 14;
	a++;
	b = sizeof(a);

	printf("a : %d\n", a);
	printf("b : %d\n", b);

	getchar();
	return 0;
}
```

EXP-08. 포인터 연산이 정확하게 수행되고 있는지 보장하라.

ㅇ 포인터 연산을 수행할 때 포인터에 더해지는 값은 자동적으로 포인터가 가리키는 데이터형으로 조정된다.

ㅇ 포인터 연산의 특징

1) 주소상수 + 정수형상수(n)

주소상수 + n 은 주소상수 + (n * 주소상수가 가리키곳의 데이터형의 크기)

2) 주소상수 - 주소상수

주소에 해당하는 기억공간 간의 첨자(index)차이가 계산 됨

[문제코드]

```c
#include <stdio.h>
#include <stdio_ext.h>
#include <string.h>

#define BUFFER_SIZE 5
int getNumber(int *);

int main()
{
        int i;
        int p;
        int buffer[BUFFER_SIZE];
        int *bufptr = buffer;

        while (bufptr < (buffer + sizeof(buffer)))
        {
                if (getNumber(&p))
                {
                        *bufptr++ = p;
                }
        }

        for (i=0 ; i<BUFFER_SIZE ; i++)
        {
                printf("buffer[%d] : %d\n", i, *(buffer+i));
        }

        __fpurge(stdin);
        getchar();
        return 0;
}
```

```
int getNumber(int *ptr)
{
        char tmp[10];
        char *nr, *br;

        *ptr = 0;
        memset(tmp, 0, sizeof(tmp));
        printf("3자리 정수입력 : ");
        while (1)
        {
                if (fgets(tmp, sizeof(tmp), stdin) == NULL)
                {
                        return 0;
                }
                // fgets()함수로 받아들인 문자열에 개행문자가 있다고 가정하지
마라.
                nr = strchr(tmp, '\n');
                *nr = nr ? '\0' : *nr;
                if (strlen(tmp) == 0 || strlen(tmp) > 3)
                {
                        return 0;
                }

                // 문자가 없거나 중간에 숫자가 아닌 문자가 존재하는가?
                *ptr = (int)strtol(tmp, &br, 10);
                if (br == tmp || *br != '\0')
                {
                        return 0;
                }

                break;
        }
        return 1;
}
```

[해결방법]

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

#define BUFFER_SIZE 5
int getNumber(int *);

int main()
{
        int i;
        int p;
        int buffer[BUFFER_SIZE];
        int *bufptr = buffer;

        while (bufptr < (buffer + BUFFER_SIZE))
```

```
        {
                if (getNumber(&p))
                {
                        *bufptr++ = p;
                }
        }

        for (i=0 ; i<BUFFER_SIZE ; i++)
        {
                printf("buffer[%d] : %d\n", i, *(buffer+i));
        }

        __fpurge(stdin);
        getchar();
        return 0;
}

int getNumber(int *ptr)
{
        char tmp[10];
        char *nr, *br;

        *ptr = 0;
        memset(tmp, 0, sizeof(tmp));
        printf("3자리 정수입력 : ");
        while (1)
        {
                if (fgets(tmp, sizeof(tmp), stdin) == NULL)
                {
                        return 0;
                }
                // fgets()함수로 받아들인 문자열에 개행문자가 있다고 가정하지
마라.
                nr = strchr(tmp, '\n');
                *nr = nr ? '\0' : *nr;
                if (strlen(tmp) == 0 || strlen(tmp) > 3)
                {
                        return 0;
                }

                // 문자가 없거나 중간에 숫자가 아닌 문자가 존재하는가?
                *ptr = (int)strtol(tmp, &br, 10);
                if (br == tmp || *br != '\0')
                {
                        return 0;
                }

                break;
        }
        return 1;
}
```

EXP-10. 하위 표현식의 평가 순서나 부수효과가 발생할 수 있는 영역의 순서에 의존하지 마라.

하위 표현식의 평가나 부수효과가 발생하는 순서가 지정되어 있지 않은 경우

- 함수에 주어진 인자가 평가되는 순서
- 할당문에서 피연산자가 평가되는 순서
- 초기화 표현식에서 나열된 객체들이 부소 효과를 갖는 순서
  (객체의 초기화 순서가 나열 순서와 같을 필요가 없다.)

다음 코드의 실행결과, extern변수 g에 1이 할당될 가능성과 2가 할당될 가능성은 반반이다.

[문제코드]

```
#include <stdio.h>

int g;

int f(int i)
{
        g = i;
        return i;
}

int main()
{
        int x = f(1) + f(2);

        printf("extern g의 값 : %d\n", g);

        getchar();
        return 0;
}
```

[해결방법]

```
#include <stdio.h>

int g;

int f(int i)
{
        g = i;
        return i;
```

```
}

int main()
{
        int x = f(1);
        x += f(2);

        printf("extern g의 값 : %d\n", g);

        getchar();
        return 0;
}
```

EXP-11. 호환되지 않는 타입들에는 연산자를 적용하지 마라.

비트필드 구조에 맴버가 할당되는 순서는 컴파일러마다 차이가 있다.

[문제코드]

```
#include <stdio.h>

struct bf
{
        unsigned int m1 : 8;
        unsigned int m2 : 8;
        unsigned int m3 : 8;
        unsigned int m4 : 8;
};

int main()
{
        struct bf data;
        unsigned char *ptr;

        data.m1 = 0;
        data.m2 = 0;
        data.m3 = 0;
        data.m4 = 0;
        ptr = (unsigned int)&data;
        (*ptr)++;

        printf("data.m1 : %u\n", data.m1);
        printf("data.m1 : %u\n", data.m2);
        printf("data.m1 : %u\n", data.m3);
        printf("data.m1 : %u\n", data.m4);

        getchar();
        return 0;
}
```

[해결방법] 수정할 필드를 명백히 지정한다.

```
#include <stdio.h>

struct bf
{
        unsigned int m1 : 8;
        unsigned int m2 : 8;
        unsigned int m3 : 8;
        unsigned int m4 : 8;
};

int main()
{
        struct bf data;

        data.m1 = 0;
        data.m2 = 0;
        data.m3 = 0;
        data.m4 = 0;
        data.m1++;      // 값을 변경할 맴버를 명확히 지정하라.

        printf("data.m1 : %u\n", data.m1);
        printf("data.m1 : %u\n", data.m2);
        printf("data.m1 : %u\n", data.m3);
        printf("data.m1 : %u\n", data.m4);

        getchar();
        return 0;
}
```

EXP-12. 함수에 의해 반환되는 값을 무시하지 마라.

함수는 반환 시 유용한 값을 전달하는데, 대게 이 값은 함수가 작업을 성공 또는 실패 했는지를 판단하는데 사용된다.

[문제코드] scanf() 함수가 정상적으로 데이터를 입력받았는지 확인하지 않고 있다.

```
#include <stdio.h>

void myflush();

int main()
{
        int age;

        printf("연령입력 : ");
```

```
        scanf("%d", &age);
        myflush();
        printf("입력된 연령 : %d\n", age);

        getchar();
        return 0;
}

void myflush()
{
        while (getchar() != '\n'){}
}
```

[해결방법] scanf()함수의 리턴값을 검사한다.

```
#include <stdio.h>

void myflush();

int main()
{
        int res, age;

        printf("연령입력 : ");
        res = scanf("%d", &age);
        myflush();

        if (res != 1)
        {
                printf("입력오류입니다.\n");
        }
        else
        {
                printf("입력된 연령 : %d\n", age);
        }

        getchar();
        return 0;
}

void myflush()
{
        while (getchar() != '\n'){}
}
```

EXP-31. 어썰션의 부수효과를 피하라.

○ 표준 assert 매크로와 함께 사용되는 표현식은 부수 효과를 가지면 안된다.

[부적절한 코드] assert()함수 호출시 인자에 대하여 연산을 수행하고 있다.

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

void myflush();
void process(int i);

int main()
{
        int res, i;

        printf("정수입력 : ");
        if (scanf("%d", &i) != 1)
        {
                printf("입력오류발생!\n");
        }
        else
        {
                process(i);
        }

        myflush();
        getchar();
        return 0;
}

void process(int i)
{
        assert(i++ > 0);
        printf("i : %d\n", i);
}

void myflush()
{
        while (getchar() != '\n');
}
```

[해결방법]

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

void myflush();
void process(int i);

int main()
{
	int res, i;

	printf("정수입력 : ");
	if (scanf("%d", &i) != 1)
	{
		printf("입력오류발생!\n");
	}
	else
	{
		process(i);
	}

	myflush();
	getchar();
	return 0;
}

void process(int i)
{
	assert(i > 0);
	i++;
	printf("i : %d\n", i);
}

void myflush()
{
	while (getchar() != '\n');
}
```

EXP-33. 초기화 되지 않은 메모리를 참조하지 마라.

○ auto 변수는 초기화 되기 전에는 gargage 데이터가 들어있다.

○ malloc()함수로 동적 할당된 메모리 공간에도 정해지지 않은 값들이 들어있기 때문에 초기화 하기 전에 사용하면 예상하지 못한 결과를 초래한다.

☞ 컴파일러는 초기화되지 않은 변수의 주소가 함수에 전달될 때, 변수가 함수내에서 초기화된다고 가정한다. 이렇게 컴파일러는 변수 초기화가 실패해도 진단하지 못하므로 프로그래머가 정확성을 위해 부가적으로 점검하여야 한다.

[문제코드]

```
#include <stdio.h>
#include <stdio_ext.h>
#include <malloc.h>
#include <string.h>
#define BUF_NAME_SIZE 100

void getName(char *);

int main()
{
        char *name;
        getName(name);

        printf("입력된 이름 : %s\n", name);

        __fpurge(stdin);
        getchar();
        return 0;
}

void getName(char *name)
{
        char *nr = NULL;
        name = (char *)malloc(BUF_NAME_SIZE);
        if (name == NULL)
        {
                printf("동적메모리 할당실패!!!\n");
                return;
        }

        printf("이름입력 : ");
        fgets(name, BUF_NAME_SIZE, stdin);
        nr = strchr(name, '\n');
        if (nr != NULL) *nr = '\0';
```

```
        return;
}
```

[해결방법 1]

```
#include <stdio.h>
#include <stdio_ext.h>
#include <malloc.h>
#include <string.h>
#define BUF_NAME_SIZE 100

char * getName();

int main()
{
        char *name;
        name = getName();

        printf("입력된 이름 : %s\n", name);

        __fpurge(stdin);
        getchar();
        return 0;
}

char * getName()
{
        char *name = NULL;
        char *nr = NULL;

        name = (char *)malloc(BUF_NAME_SIZE);
        if (name == NULL)
        {
                printf("동적메모리 할당실패!!!\n");
                return name;
        }

        printf("이름입력 : ");
        fgets(name, BUF_NAME_SIZE, stdin);
        nr = strchr(name, '\n');
        if (nr != NULL) *nr = '\0';

        return name;
}
```

[해결방법 2]

```
#include <stdio.h>
#include <stdio_ext.h>
#include <malloc.h>
#include <string.h>
#define BUF_NAME_SIZE 100

void getName(char **);

int main()
{
        char *name;
        getName(&name);

        printf("입력된 이름 : %s\n", name);

        __fpurge(stdin);
        getchar();
        return 0;
}

void getName(char **name)
{
        char *nr = NULL;
        *name = (char *)malloc(BUF_NAME_SIZE);
        if (name == NULL)
        {
                printf("동적메모리 할당실패!!!\n");
                return;
        }

        printf("이름입력 : ");
        fgets(*name, BUF_NAME_SIZE, stdin);
        nr = strchr(*name, '\n');
        if (nr != NULL) *nr = '\0';

        return;
}
```

EXP-34. 널 포인터가 역참조 되지 않음을 보장하라.

NULL 포인터를 역참조 하면 프로그램이 알 수 없는 상태가 되는데, 일반적으로는 종료된다.

[문제코드] 동적메모리 할당 후, 할당 성공여부를 확인하지 않고 있다.

```
#include <stdio.h>
#include <malloc.h>
#include <string.h>
#define BUF_NAME_SIZE 100

int main()
{
        char *name;
        char *nr;

        name = (char *)malloc(BUF_NAME_SIZE);

        printf("이름입력 : ");
        fgets(name, BUF_NAME_SIZE, stdin);
        nr = strchr(name, '\n');
        if (nr != NULL) *nr = '\0';
        printf("입력된 이름 : %s\n", name);

        free(name);
        name = NULL;
        getchar();
        return 0;
}
```

[해결방법] malloc()함수의 반환값이 NULL이 아님을 보장해준다.

```
#include <stdio.h>
#include <malloc.h>
#include <string.h>
#define BUF_NAME_SIZE 100

int main()
{
        char *name;
        char *nr;

        name = (char *)malloc(BUF_NAME_SIZE);
        if (name == NULL)
        {
                printf("동적메모리 할당 실패!!!\n");
                return 1;
        }
```

```
        printf("이름입력 : ");
        fgets(name, BUF_NAME_SIZE, stdin);
        nr = strchr(name, '\n');
        if (nr != NULL) *nr = '\0';
        printf("입력된 이름 : %s\n", name);

        free(name);
        name = NULL;
        getchar();
        return 0;
}
```

PART 2. 정수, 부동소수점, 배열, 문자와 문자열

- ▶ 정수 변환 규칙의 이해
- ▶ 정수 관련 변수, 함수 및 연산자 고려사항
- ▶ 부동소수점 관련 변수, 함수 및 연산자 고려사항
- ▶ 배열의 선언 및 초기화, 사용에 대한 고려사항
- ▶ 문자와 문자열의 선언 및 초기화, 사용에 대한 고려사항

제5장 정수

INT-00. 플랫폼에 따른 데이터 모델을 이해하고 있어라.

데이터 모델은 표준 데이터 타입에 대한 할당되는 크기를 정의한다. 사용하는 플랫폼에 따른 데이터 모델을 이해하는 일은 중요하다.

타입	iAPX68	IA32	IA64	SPARK64	ARM32	Alpha	64bit POSIX
char	8	8	8	8	8	8	8
short	16	16	16	16	16	16	16
int	16	32	32	32	32	32	32
long	32	32	32	64	32	64	64
long long	N/A	64	64	64	64	64	64
포인터	16/32	32	64	64	32	64	64

☞ limits.h 파일에는 표준 정수 타입의 범위를 결정하는 매크로가 지정되어 있다.

UINT_MAX : unsigned int형이 가질 수 있는 최대 값

LONG_MIN : long int가 가질 수 있는 최소 값

☞ stdint.h 파일에는 데이터 모델에 종속되지 않고 사용할 수 있는 특정 크기로 제한된 타입들이 있다.

int_least_32_t : 플랫폼에서 지원하는 최소 32비트 이상의 signed형 정수타입

uint_fast16_t : 최소 16비트 이상을 갖는 unsigned 정수 타입

[문제코드] unsigned int값 두 개를 곱할 때 64비트 리눅스 시스템에서는 정상 동작 하지만 윈도우 시스템에서는 문제가 발생한다.

```c
#include <stdio.h>

int main()
{
        unsigned int a, b;
        unsigned long c;

        a = 1000000000;
        b = 100;
        c = (unsigned long)a * b;

        printf("c = %lu\n", c);
```

```
        getchar();
        return 0;
}
```

[해결방법] 결과를 보존할 수 있는 큰 unsigned int 타입을 사용

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

int main()
{
        unsigned int a, b;
        uintmax_t c;

        a = 1000000000;
        b = 100;
        c = (uintmax_t)a * b;

        printf("c = %llu\n", c);

        getchar();
        return 0;
}
```

INT-01. 배열의 크기를 나타내는 값이나 루프 카운터로 사용되어지는 변수의 데이터 형은 unsigned 정수형 또는 size_t 형을 사용하라.

[생각해볼 코드]

```c
#include<stdio.h>

int main()
{
        char  i;

        for(i=0; i<100; i--)
        {
                printf("i=%d\n", i);
        }
        getchar();

        for(i=0; i<200; i++)
        {
                printf("i=%d\n", i);
        }
        getchar();

        return 0;
}
```

[생각해볼 코드]

```c
#include<stdio.h>

int main()
{
        char  x = 127;
        char  y =  x + 1;
        printf("x = %d   y = %d\n", x, y);

        x = - 128;
        y = x - 1;
        printf("x = %d   y = %d\n", x, y);

        getchar();
        return 0;
}
```

size_t 타입은 sizeof연산의 결과로 얻어지는 unsigned 정수 타입이다. size_t의 한계값은 SIZE_MAX 매크로로 지정되어 있다.

[문제코드] copy()함수의 루프카운터로 사용된 변수 i는 signed int 형으로 n>INT_MAX 인 값에 대해서 오버플로우가 발생한다.

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

char *copy(const char *str, size_t n)
{
        int i;
        char *p;
        p = (char *)malloc(n * sizeof(char));
        if (p == NULL)
        {
                printf("동적할당오류!!!\n");
                return NULL;
        }

        for (i=0 ; i<n ; i++)
        {
                p[i] = *str++;
        }
        return p;
}

int main()
{
        char *message;
        char string[] = "문자열을 목사합니다.";

        if ((message = copy(string, sizeof(string))) != NULL)
        {
                printf("원본 : %s\n", string);
                printf("사본 : %s\n", message);
        }
        else
        {
                printf("문자열복사실패!!!\n");
                getchar();
                return 1;
        }

        getchar();
        return 0;
}
```

[해결방법] i를 size_t 타입으로 수정한다.

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

char *copy(const char *str, size_t n)
{
        size_t i;
        char *p;
        p = (char *)malloc(n * sizeof(char));
        if (p == NULL)
        {
                printf("동적할당오류!!!\n");
                return NULL;
        }

        for (i=0 ; i<n ; i++)
        {
                p[i] = *str++;
        }
        return p;
}

int main()
{
        char *message;
        char string[] = "문자열을 목사합니다.";

        if ((message = copy(string, sizeof(string))) != NULL)
        {
                printf("원본 : %s\n", string);
                printf("사본 : %s\n", message);
        }
        else
        {
                printf("문자열복사실패!!!\n");
                getchar();
                return 1;
        }

        getchar();
        return 0;
}
```

INT-02. 정수 변환 규칙을 이해하라.

▶ 정수변환

- 캐스팅을 통한 명시적인 변환
- 연산식에 의한 묵시적인 변환

▶ 정수변환 규칙

1. 정수의 승계
2. 정수 변환 순위
3. 일반적인 산술변환

1) 정수의 승계

int보다 작은 정수형은 연산이 수행될 때 int형나 unsigned int형으로 변환되어 연산된다. 정수의 승계는 연산의 중간에 사용되는 값의 오버플로우를 예방하기 위해서 수행된다.

```c
#include <stdio.h>

int main()
{
        signed char result, c1, c2, c3;

        c1 = 100;
        c2 = 3;
        c3 = 4;

        /* 300 / 4 *의 결과 값이
        result에 signed char형으로 변환되어 저장된다. */
        result = c1 * c2 / c3;
        printf("result : %d\n", result);

        getchar();
        return 0;
}
```

2) 정수 변환 순위

비트수가 많은 자료형이 높은 순위를 갖는다. 정수 변환 순위는 일반적인 산술변환에서 각기 다른 정수 타입에 대한 연산이 수행될 때, 어떤 변환이 필요한지를 결정하는데 사용한다.

ㅇ 각기 다른 두 signed 정수형는 순위가 다르다.

ㅇ signed 정수형의 변환 순위는 자신보다 정밀도가 작은 다른 signed 정수형보다 순위가 높다.

```
long long int > long int > int > short > char
```

☞ 모든 unsigned 정수형 순위는 일치하는 signed 정수형의 순위와 같다.

3) 일반적인 산술변환

두 피연산자의 자료형을 일치시켜야 하는 경우에 적용하는 일종의 규칙

ㅇ 이항 연산 시 두 피연산자가 같은 자료형으로 변환 된다.

ㅇ 조건 연산자 (?:)의 두번째와 세번째 피연산자는 같은 자료형으로 변환된다.

- 정수의 승계가 먼저 진행된다.
- 두 개의 피연산자가 같은 자료형이면 변환하지 않는다.
- unsigned 정수형 피연산자의 자료형이 다른 피연산자의 자료형의 순위보다 크거가 같거나 큰 경우, signed 정수형의 피연산자의 데이터형이 unsigned 정수형의 피연산자 데이터형으로 변환된다.
- signed 정수형 피연산자의 자료형이 다른 피연산자의 자료형보다 순위보다 큰 경우, signed 정수형 피연산자의 자료형이 다른 정수형 피연산자의 값을 표현할 수 있다면 signed 정수형 피연산자의 자료형으로 변환된다. 반면 signed 정수형 피연산자의 자료형이 다른 정수형 피연산자의 값을 표현 할 수 없다면 signed 정수형 피연산자와 일치하는 unsigned 정수형으로 두 피연산자가 모두 변환된다.

[정수변환규칙의 예]

char : 8bit / int : 32bit / long long 64bit 시스템에서

```
signed char sc = SCHAR_MAX;
unsigned char uc = UCHAR_MAX;
signed long long all = sc + uc;
```

(1) 정수의 승계에 의해 sc, uc 모두 int로 변환된다.

signed long long = int;

(2) int의 결과에서 sign부분만 64비트 영역으로 확장된다.

※ 안전하게 수행 됨

[정수 변환의 예]

char : 32bit / int : 32bit / long long : 64bit 시스템에서

```
signed char sc = SCHAR_MAX;
unsigned char uc = UCHAR_MAX;
signed long long all = sc + uc;
```

(1) sc는 int형으로, uc는 unsigned int형으로 변환된다.

signed long long = signed int + unsigned int

(2) signed int가 unsigned int형으로 변환된다.

signed long long = unsigned int + unsigned int
※ 이때 unsigned int + unsigned int 에서 오버플로우가 발생한다.

(3) 결과값이 signed long long 형으로 변환되어 저장 된다.

[생각해볼 예제]

```c
#include <stdio.h>

int main()
{
	int x = -1;
	unsigned int y = 1;

	if (x > y)
	{
		printf("True\n");
	}
	else
	{
		printf("False\n");
	}

	getchar();
	return 0;
}
```

[생각해볼 예제]

```c
#include <stdio.h>

int main()
{
	char x = -1;
	unsigned char y = 1;

	if (x > y)
	{
		printf("True\n");
	}
	else
	{
		printf("False\n");
	}

	getchar();
	return 0;
}
```

[문제코드] 서로 다른 타입의 연산을 수행할 때 주의해야 한다.

```c
#include <stdio.h>

int main()
{
	int si = -1;
	unsigned int ui = 1;

	printf("%d\n", si < ui);

	getchar();
	return 0;
}
```

[문제해결] signed int 값을 사용해 비교하도록 강제형변환을 한다.

```c
#include <stdio.h>

int main()
{
	int si = -1;
	unsigned int ui = 1;

	printf("%d\n", si < (int)ui);

	getchar();
	return 0;
}
```

INT-04. 불분명한 소스에서 얻어지는 정수의 값은 제한을 강제하라.

신뢰할 수 없는 소스로 부터 얻어지는 정수 값은 식별할 수 있는 상한값과 하한값이 있는지 확인하기 위해 반드시 평가되어야 한다.

아래 코드에서 table의 길이를 결정하는데 쓰이는 length의 값을 신뢰할 수 없다면 malloc() 호출 실패등의 문제가 발생할 수 있으므로 length의 적용가능한 범위를 평가하여야 한다.

[문제코드]

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

enum { MAX_TABLE_LENGTH = 256 };

int create_table(size_t);
void myflush();

int main()
{
        int length;

        while (1)
        {
                printf("문자열을 수 : ");
                if (scanf("%d", &length) != 1 || length < 0)
                {
                        myflush();
                        printf("입력오류");
                        continue;
                }

                create_table(length);
                break;
        }

        myflush();
        getchar();
        return 0;
}

int create_table(size_t length)
{
        size_t table_length;
        char **table;

        table_length = length * sizeof(char *);
        table = (char **)malloc(table_length);
        if (table == NULL)
        {
                printf("동적메모리할당실패!!!\n");
                return 0;
```

```
        }

        printf("동적메모리할당성공!!!\n");
        /* table 을 사용하는 코드 */
        free(table);
        return 1;
}

void myflush()
{
        while (getchar() != '\n');
}
```

[해결방법]

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

enum { MAX_TABLE_LENGTH = 256 };

int create_table(size_t);
void myflush();

int main()
{
        int length;

        while (1)
        {
                printf("문자열을 수 : ");
                if (scanf("%d", &length) != 1 || length < 0)
                {
                        myflush();
                        printf("입력오류");
                        continue;
                }

                create_table(length);
                break;
        }

        myflush();
        getchar();
        return 0;
}

int create_table(size_t length)
{
        size_t table_length;
        char **table;

        if (length == 0 || length > MAX_TABLE_LENGTH)
```

```
        {
                printf("유효하지 않은 수 입니다.\n");
                return 0;
        }

        table_length = length * sizeof(char *);
        table = (char **)malloc(table_length);
        if (table == NULL)
        {
                printf("동적메모리할당실패!!!\n");
                return 0;
        }

        printf("동적메모리할당성공!!!\n");
        /* table 을 사용하는 코드 */
        free(table);
        return 1;
}

void myflush()
{
        while (getchar() != '\n');
}
```

INT-05. 모든 가능한 입력을 처리할 수 없다면 문자 데이터 변환을 위해 입력함수를 사용하지 마라.

문자열을 읽어 정수나 부동소수점 수로 변환하여 저장하는 함수는 인자 타입으로 표현 불가능한 숫자를 담을 수 없으므로 사용하지 않는 것이 좋다.

[문제코드] scanf()함수는 입력 문자열을 정수로 변환할 수 없는 경우, 정의되지 않은 행동을 유발한다.

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

int main()
{
        long s1;
        int res;

        printf("long형 정수입력 : ");
        res = scanf("%ld", &s1);
        if (res != 1)
        {
                printf("데이터 입력 오류가 발생하였습니다.\n");
        }

        printf("입력된 데이터 : %ld\n", s1);
        printf("scanf()함수 실행결과 : %d\n", res);

        __fpurge(stdin);
        getchar();
        return 0;
}
```

[해결방법] 입력 문자열은 fgets()함수를 사용하여 입력하고, 문자열을 정수로 변환하기 위해 strtoul()함수를 사용한다. strtoul()함수는 입력 값이 long영역에서 유효한지를 점검하는 에러 체크 메카니즘을 제공한다.

```
#include <stdio.h>
#include <stdio_ext.h>
#include <stdlib.h>
#include <limits.h>
#include <errno.h>

int main()
{
        long s1;
        char buff[25];
        char *endptr;

        fgets(buff, sizeof(buff), stdin);

        errno = 0;
        s1 = strtoul(buff, &endptr, 10);

        if (errno == ERANGE)
        {
                printf("ULONG_MAX보다 큰 값이 입력되었습니다.\n");
        }
        else if (endptr == buff && *endptr == '\n')
        {
                printf("데이터가 입력되지 않았습니다.\n");
        }
        else if (*endptr != '\n' && *endptr != '\0')
        {
                printf("숫자로 변환할 수 없는 문자가 포함되어 있습니다.\n");
        }
        else
        {
                printf("입력된 데이터 : %ld\n", s1);
        }

        __fpurge(stdin);
        getchar();
        return 0;
}
```

INT-06. 문자열 토큰을 정수로 변환할 때에는 strtol()이나 관련 함수를 사용하라.

문자열 토큰을 정수로 변환할 때에는 strtol()이나 관련 함수를 사용하라. 아래의 함수들은 범위 체크 등 신뢰성 있는 에러 처리를 제공한다.

```
strtol(), strtoll(), strtoul(), strtoull(), strtod(), strtof()
```

위의 함수들은 다음과 같은 기능을 제공한다.

- 에러 발생 시 errno를 설정한다.
- 문자열에 정수 값이 없을 경우, 리턴되는 0과 실제 0이 입력된 경우를 변환 에러 발생 위치의 확인을 통해 확인이 가능하다.

아래의 함수들은 부적절한 입력에 대한 에러를 출력해주는 메커니즘이 부실하므로 사용하지 않는것이 좋다.

```
atoi(), atol(), atof(), sscanf()
```

INT-07. 숫자를 저장할 char형 변수에는 명시적으로 singed나 unsigned를 지정하라.

char, signed char, unsigned char를 통틀어서 문자형(character type)이라고 한다. char형 변수에 저장되는 값은 문자가 아닌 해당 문자의 ASCII 코드값이 저장된다.

[문제코드] char형 변수 c는 signed형일 수도 있고, unsigned형일 수도 있다.

```c
#include <stdio.h>

int main()
{
	char c = 200;
	int i = 1000;

	printf("i/c = %d\n", i/c);

	getchar();
	return 0;
}
```

[해결방법] signed 또는 unsigned형임을 명확히 명시한다.

```c
#include <stdio.h>

int main()
{
	unsigned char c = 200;
	int i = 1000;

	printf("i/c = %d\n", i/c);

	getchar();
	return 0;
}
```

INT-09. 열거형 상수가 유일한 값으로 매핑되도록 보장하라.

C의 열거형은 정수형으로 매핑된다. 일반적으로 각 열거형 상수는 고유한 개별의 값(매핑하지 않는 경우 첫번째 맴버의 값이 0)으로 매핑된다고 생각하지만, 열거형 타입 맴버들이 서로 같은 값을 갖는 명확하지 않은 에러가 종종 만들어지기도 한다.

아래의 코드에서는 열거형 맴버의 값이 명시적으로 매핑되고 있으나 값이 중복되는 문제가 발생한다. 이로인해 yellow와 indigo가 green과 violet가 같은 상수값을 지니게 된다. 그러므로 yellow와 indigo 그리고 green과 violet는 사용함에 있어서 제약이 따르게 된다.

switch ~ case 구문에서의 case의 상수값은 중복될 수 없다.

```
enum { red=4, orange, yellow, green, indigo=6, violet };
```

열거형 타입의 선언은 반드시 아래의 내용을 따르도록 하라.

1. 명시적인 선언을 하지 않는다.

```
enum { red, orange, yellow, green, indigo, violet };
```

2. 첫 번째 맴버에 대해서만 값을 지정한다.

```
enum { red=4, orange, yellow, green, indigo, violet };
```

3. 모든 맴버에 대하여 명시적으로 값을 지정한다.

```
enum { red=4, orange=5, yellow=6, green=7, indigo=6, violet=7 };
```

INT-10. % 연산자를 쓸 때 나머지가 양수라고 가정하지 마라.

○ C89에서는 음수 피연산자에 대한 나머지 연산의 의미가 구현마다 다르게 정의된다.

○ 여러 플랫폼의 다양한 컴파일 환경을 가진 경우, 개발자는 % 연산자의 동작을 표준에 의지할 수는 없다.

○ C99에서 결과의 부호는 피제수(죄측의 피연산자)의 부호를 따른다.

[문제코드] 모듈로 형식의 수식의 결과로 배열의 index를 지정하는데 있어서 피제수가 int 형이므로 항상 양수임을 보장할 수가 없다.

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

int insert(int, int *, int, int);

int main()
{
        int i;
        int index;
        int value;
        int ary[5] = {};
        int size = sizeof(ary)/sizeof(ary[0]);

        while (1)
        {
                printf("입력할 위치 : ");
                __fpurge(stdin);
                if (scanf("%d", &index) != 1)
                {
                        printf("데이터입력오류!!!\n");
                        return 1;
                }

                printf("입력할 데이터 : ");
                __fpurge(stdin);
                if (scanf("%d", &value) != 1)
                {
                        printf("데이터입력오류!!!\n");
                        return 1;
                }

                index = insert(index, ary, size, value);
                printf("%d번째 방의 값을 초기화 하였습니다.\n", index);
                for (i=0 ; i<size ; i++)
                {
                        printf("ary[%d] : %d\t", i, ary[i]);
                }
```

```
                printf("\n");
        }

        __fpurge(stdin);
        getchar();
        return 0;
}

int insert(int index, int *list, int size, int value)
{
        if (size != 0)
        {
                index = index % size;
                list[index] = value;
                return index;
        }
        else
        {
                return -1;
        }
}
```

[해결방법]

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

int insert(size_t, int *, int, int);

int main()
{
        int i;
        int index;
        int value;
        int ary[5] = {};
        int size = sizeof(ary)/sizeof(ary[0]);

        while (1)
        {
                printf("입력할 위치 : ");
                __fpurge(stdin);
                if (scanf("%d", &index) != 1)
                {
                        printf("데이터입력오류!!!\n");
                        return 1;
                }

                printf("입력할 데이터 : ");
                __fpurge(stdin);
                if (scanf("%d", &value) != 1)
                {
                        printf("데이터입력오류!!!\n");
```

```
                        return 1;
                }

                index = insert(index, ary, size, value);
                printf("%d번째 방의 값을 초기화 하였습니다.\n", index);
                for (i=0 ; i<size ; i++)
                {
                        printf("ary[%d] : %d\t", i, ary[i]);
                }
                printf("\n");
        }

        __fpurge(stdin);
        getchar();
        return 0;
}

int insert(size_t index, int *list, int size, int value)
{
        if (size != 0)
        {
                index = index % size;
                list[index] = value;
                return index;
        }
        else
        {
                return -1;
        }
}
```

INT-12. 표현식에서 signed, unsigned 표시가 없는 int비트 필드의 타입을 가정하지 마라.

[문제코드] 비트필드에서 int형 비트필드를 지정하면 signed int형인지 unsigned int형인지 모호해 진다.

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

struct {
        int a : 8;
} bits = {255};

int main()
{
        printf("bits.a : %d\n", bits.a);

        __fpurge(stdin);
        getchar();
        return 0;
}
```

[해결방법]

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

struct {
        unsigned int a : 8;
} bits = {255};

int main()
{
        printf("bits.a : %d\n", bits.a);

        __fpurge(stdin);
        getchar();
        return 0;
}
```

INT-13. 비트 연산자는 unsigned 피연산자에만 사용하라.

비트연산자(!, >>, <<, &, |, ^)는 signed 정수에 대한 비트연산이 구현마다 다르게 정의되어 있으므로 unsigned 정수 피연산자에 대해서만 사용하도록 하자.

[문제코드] signed 정수형에 대한 >> 연산은 부호값이 오른쪽에 채워지게 된다.

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

void charPrinter(char);

int main()
{
        char ch = CHAR_MIN;

        charPrinter(ch);
        ch >>= 1;
        charPrinter(ch);

        __fpurge(stdin);
        getchar();
        return 0;
}

void charPrinter(char v)
{
        unsigned char op = 1;
        op <<= sizeof(char) * CHAR_BIT - 1;

        printf("%10d : ", v);
        while (op > 0)
        {
                if (v & op) printf("1");
                else printf("0");
                op >>= 1;
        }
        printf("\n");
}
```

[해결방법]

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

void charPrinter(char);

int main()
{
```

```
        unsigned char ch = CHAR_MAX;

        charPrinter(ch);
        ch >>= 1;
        charPrinter(ch);

        __fpurge(stdin);
        getchar();
        return 0;
}

void charPrinter(char v)
{
        unsigned char op = 1;
        op <<= sizeof(char) * CHAR_BIT - 1;

        printf("%10d : ", v);
        while (op > 0)
        {
                if (v & op) printf("1");
                else printf("0");
                op >>= 1;
        }
        printf("\n");
}
```

INT-30. unsigned 정수 연산이 래핑되지 않도록 주의하라.

ㅇ unsigned 피 연산자를 사용한 계산은 결코 오버플로우가 발생하지 않는다. 연산 결과 값이 저장될 정수 타입으로 표현할 수 없는 경우, 나머지(%)연산으로 값을 줄여(Wrap around) 표현하기 때문이다. 이것을 정수 래핑(Wrapping)현상이라고 한다.

ㅇ 신뢰할 수 없는 소스로부터 얻어진 정수 값이 아래와 같은 곳에 사용된다면 절대 래핑을 허용하여서는 안된다.

- 배열의 인덱스
- 포인터 연산의 일부
- 루프 카운터
- 메모리 할당 함수의 인자
- 그 밖의 보안에 민감한 코드

[문제코드] unsigned 피 연산자 끼리의 덧셈 과정에서 unsigned 정수 래핑이 발생한다.

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

int main()
{
        unsigned int ui1, ui2, sum = 0;

        ui1 = UINT_MAX;
        ui2 = INT_MAX;

        sum = ui1 + ui2;

        printf("sum : %ld\n", sum);

        getchar();
        return 0;
}
```

[해결방법]

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

int main()
{
	unsigned int ui1, ui2, sum = 0;

	ui1 = UINT_MAX;
	ui2 = INT_MAX;

	if (UINT_MAX - ui1 < ui2)
	{
		printf("래핑이 발생합니다.!!!\n");
		/* 래핑관련에러처리 */
	}
	else
	{
		sum = ui1 + ui2;
	}

	printf("sum : %ld\n", sum);

	getchar();
	return 0;
}
```

INT-34. 피연산자의 비트보다 더 많은 비트를 시프트하지 마라.

아래의 예제에서는 좌측 피연산자의 비트수 보다 더 많이 쉬프트를 하고 있다.
그러므로 아래 예제의 쉬프트 연산은 다음과 같이 수행될 것이다.

```
        result = ui1 << ui2;
▶       result = ui1 << (ui2 % ui1의비트수);
```

[문제코드] 피연산자의 비트수 보다 더 많은 비트를 shift하려고 한다.

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

int main()
{
        unsigned int ui1, ui2, uresult = 0;

        ui1 = 1;
        ui2 = 31;
        uresult = ui1 << ui2;
        printf("uresult : %u\n", uresult);

        getchar();
        return 0;
}
```

[해결방법]

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

int main()
{
        unsigned int ui1, ui2, uresult = 0;

        ui1 = 1;
        ui2 = 31;

        if (ui2 >= sizeof(ui1) * CHAR_BIT)
        {
                printf("피연산자의 비트수 보다 더 많이 이동시킬 수
없습니다!!!\n");
        }
        uresult = ui1 << ui2;
        printf("uresult : %u\n", uresult);

        getchar();
```

```
        return 0;
}
```

[문제코드] unsigned 피연산자 끼리의 left shift 연산과정에서 unsigned 정수 래핑이 발생

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

int main()
{
        unsigned int ui1, ui2, uresult = 0;

        ui1 = 256;
        ui2 = 28;
        uresult = ui1 << ui2;
        printf("uresult : %u\n", uresult);

        getchar();
        return 0;
}
```

[해결방법]

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

int main()
{
        unsigned int ui1, ui2, uresult = 0;

        ui1 = 256;
        ui2 = 28;

        if (ui2 > (UINT_MAX >> ui2))
        {
                printf("래핑이 발생합니다.\n");
                /* 래핑에 대한 에러 처리 */
        }
        else
        {
                uresult = ui1 << ui2;
        }

        printf("uresult : %u\n", uresult);
        getchar();
        return 0;
}
```

제6장 부동소수점 (FLP)

FLP-00. 부동소수점의 저장방식을 이해하라.

○ IEEE754 표준에서 정의하고 있는 방식이 가장 일반적인 부동소수점 방식이다.

○ 그러나 IBM의 부동소수점 표기방식도 사용되는데, 이러한 시스템들은 정밀도와 표현가능한 값의 범위가 각각 다르다.

○ 즉, 시스템별로 동일한 부동소수점의 구현사항이 보장되지 않기 때문에 정말도나 범위에 대해 어떤 가정도 하여서는 안된다.

[예제] 부동소수점의 정확도

```c
#include <stdio.h>

int main()
{
        float f = 1.0f/3.0f;
        printf("float is %.40f\n", f);
        getchar();
        return 0;
}
```

[결과]

```
gcc에서의 결과

  float is 0.3333333432674407958984375000000000000000

Dev-C에서의 결과

  float is 0.3333333432674408000000000000000000000000
```

☞ 부동소수점의 저장방식 (IEEE754 표준)

① 단정도 부동소수 (float : 32bit)

sing bit	지수부	가수부(유효숫자부)
1bit	8bit	23bit

② 배정도 부동소수 (double : 64bit)

sing bit	지수부	가수부(유효숫자부)
1bit	11bit	52bit

● sign bit : 가수부(유효숫자부)의 부호 (양수 : 0, 음수 : 1)

● 지수부 저장방식 : 지수승 + Bias값

Bias값은 지수부가 표현할 수 있는 최대 크기값을 2로 나눈 값

단정도 : (지수부 8bit에 저장할 수 있는 최대값) / 2 이므로 127

배정도 : (지수부 11bit에 저장할 수 있는 최대값) / 2 이므로 1023

● 가수부 저장방식 : 2진수로 표현된 숫자를 정규화 한 후, 소수점 이하 자리만을 저장함

정규화란? 2진수로 변환시킨 실수의 소수점의 위치를 앞에 1이 위치할 때 까지 옮기는 것을 말한다. 이때 소수점이 이동한 횟수가 지수가 되며, 소수점이 좌측으로 이동하였다면 양의 값, 우측으로 이동하였다면 음의 값이 된다.

☞ 부동소수점의 저장방식에서의 오차

① 순환소수에 의한 오차

② 유효정밀도에 의한 오차

FLP-02. 정확한 계산이 필요할 때에는 부동소수점 수를 배제할 수 있는지를 고려하라.

컴퓨터에서의 숫자의 표현은 그 크기가 제한되어 있다. 그러므로 1/3과 같은 반복되는 이진 표기값을 정확하게 표현하는 것은 대부분의 부동소수점 표현으로는 불가능하다.

[문제코드]

```
#include <stdio.h>

float mean(float *, size_t);

int main()
{
        int i;
        float array[10];
        float total;

        for (i=0 ; i<sizeof(array)/sizeof(array[0]) ; i++)
        {
                array[i] = 10.1F;
        }

        total = mean(array, sizeof(array)/sizeof(array[0]));
        printf("total = %.10f\n", total);

        getchar();
        return 0;
}

float mean(float *ary, size_t len)
{
        float total = 0.0f;
        int i;

        for (i=0 ; i<len ; i++)
        {
                total += *(ary + i);
                printf("ary[%d] : %f\t total = %f\n", i, *(ary + i),
total);
        }

        if (len != 0)
        {
                return total / len;
        }
        else
        {
                return 0.0F;
        }
}
```

[해결방법] 부동소수점수를 정수로 바꿔서 해결한다.

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

float mean(int *, size_t);

int main()
{
        int i;
        int array[10];
        float total;

        for (i=0 ; i<sizeof(array)/sizeof(array[0]) ; i++)
        {
                array[i] = 101;
        }

        total = mean(array, sizeof(array)/sizeof(array[0]));
        printf("total = %.10f\n", total);

        getchar();
        return 0;
}

float mean(int *ary, size_t len)
{
        int total = 0;
        int i;

        for (i=0 ; i<len ; i++)
        {
                total += *(ary + i);
                printf("ary[%d] : %d\t total = %d\n", i, *(ary + i),
total);
        }

        if (len != 0)
        {
                return (float)total / len;
        }
        else
        {
                return 0;
        }
}
```

FLP-30. 부동소수점 변수를 루프 카운터로 사용하지 마라.

ㅇ 부동소수점 수는 간단한 십진분수도 정확하게 표현하지 못할 수 도 있다.

ㅇ 큰 부동소수점 값에 증가 연산을 적용하면, 경우에 따라서 가능한 정밀도의 한계로 인해 값이 전혀 변하지 않을 수 도 있다.

[문제코드] 정확하게 10회 반복이 안된다. 해결방법은?

```
#include <stdio.h>

int main()
{
        float x;
        int count = 1;
        for (x=0.1f ; x<=1.0f ; x+=0.1f)
        {
                printf("%d. x = %.10f\n", count++, x);
        }

        getchar();
        return 0;
}
```

[문제코드] 자신의 정밀도로 표현이 안되는 작은 값으로 증가시켜서 값이 아예 변하지 않아 무한 반복에 빠질수도 있다.

```
#include <stdio.h>

int main()
{
        float x;
        int count = 1;

        for (x=100000001.0f ; x<=1000000010.0f ; x+=1.0f)
        {
                printf("%d. x = %.30f\n", count++, x);
        }

        getchar();
        return 0;
}
```

FLP-33. 부동소수점 연산용 정수는 먼저 부동소수점으로 바꿔라.

○ 계산 시 정수를 사용해 부동소수점 변수에 값을 할당하는 경우, 정보가 손실될 수 있다.

[문제코드] 부동소수점 변수에 저장하기 전에 정수 연산 시에 손실이 발생한다.

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

int main()
{
        short a = 533;
        int b = 6789;
        long c = 466438237;

        float d = a / 7;        // d는 76.0
        double e = b / 30;      // e는 226.0
        double f = c * 255;     // f는 오버플로우되어 음수일 수 있다.

        printf("d : %f\n", d);
        printf("d : %f\n", e);
        printf("d : %f\n", f);

        getchar();
        return 0;
}
```

[해결방법] 정수값을 부동소수점 변수에 저장하여 부동소수점으로 변환한 후, 연산수행

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

int main()
{
        short a = 533;
        int b = 6789;
        long c = 466438237;

        float d = a;
        double e = b;
        double f = c;

        d /= 7;
        e /= b;
        f *= 789;

        printf("d : %f\n", f);
        printf("e : %f\n", d);
```

```
        printf("f : %f\n", e);

        getchar();
        return 0;
}
```

FLP-34. 변환된 값을 저장할 수 있는 자료형인가를 확인하라.

부동소수점 값이 더 작은 범위나 정밀도를 가진 부동소수점 값으로 변환되거나 정수로 변환되는 경우, 혹은 정수가 부동소수점 수로 변환되는 경우, 데이터는 변환될 타입으로 표현가능한 값이어야 한다.

[문제코드] 부동소수점 수를 정수형 변수에 저장할 때, 정수부에 대한 저장을 보장받을 수 없다.

```
#include <stdio.h>

int main()
{
        float f1;
        int i1;

        f1 = 3.14E+10;
        i1 = f1;

        printf("f1 : %f\n", f1);
        printf("i1 : %d\n", i1);

        getchar();
        return 0;
}
```

[해결방법] 부동소수점의 값이 int형의 범위 안의 값인지를 검사한다.

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

int main()
{
        float f1;
        int i1;
```

```
        f1 = 3.14E+10;
        if (f1 > (float)INT_MAX || f1 < (float)(INT_MIN))
        {
                printf("int형에 저장할 수 있는 범위 밖의 수입니다!!!\n");
                /* 에러처리 */
        }
        else
        {
                i1 = f1;
                printf("f1 : %f\n", f1);
                printf("i1 : %d\n", i1);
        }

        getchar();
        return 0;
}
```

[문제코드] 변환하고자 하는 대상이 변환될 타입의 범위 밖의 값일 수 있다.

```
#include <stdio.h>

int main()
{
        float f1;
        double d1;

        d1 = 1.7E+300;
        f1 = (float)d1;
        printf("f1 : %E\n", f1);

        getchar();
        return 0;
}
```

[해결방법] 변환되는 값이 새로운 타입으로 표현될 수 있는지 확인한다.

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

int main()
{
        float f1;
        double d1;

        d1 = 1.7E+300;

        if (d1 > (double)FLT_MAX || d1 < (double)FLT_MIN)
```

```
        {
                printf("범위 밖의 데이터입니다!!!\n");
                /* 변환오류처리 */
        }
        else
        {
                f1 = (float)d1;
                printf("f1 : %E\n", f1);
        }

        getchar();
        return 0;
}
```

제7장 배열

ARR-01. 배열의 크기를 얻을 때 포인터를 sizeof연산자의 피연산자로 사용하지 마라.

- sizeof 연산자는 피연산자의 크기를 바이트 단위로 계산해준다.
- sizeof 연산자로 배열의 크기를 결정하기 위해서 사용할 때에는 주의가 필요하다.

[문제코드] 매개변수로 전달된 배열(포인터)에 대해 sizeof 연산을 수행하고 있다.

```
#include <stdio.h>

void clear(int []);

int main()
{
        size_t i;
        int array[5];

        clear(array);
        for (i=0 ; i<sizeof(array)/sizeof(array[0]) ; i++)
        {
                printf("array[%u] : %d\n", i, array[i]);
        }

        getchar();
        return 0;
}

void clear(int array[])
{
        size_t i;
        for (i=0 ; i<sizeof(array)/sizeof(array[0]) ; i++)
        {
                array[i] = 0;
        }
}
```

[해결방법] 배열이 선언된 블럭에서 크기를 계산한 후, 인자로 함께 넘긴다.

```
#include <stdio.h>

void clear(int [], size_t);

int main()
{
        size_t i;
        int array[5];
        size_t len = sizeof(array)/sizeof(array[0]);

        clear(array, len);
```

```
        for (i=0 ; i<len ; i++)
        {
                printf("array[%u] : %d\n", i, array[i]);
        }

        getchar();
        return 0;
}

void clear(int array[], size_t len)
{
        size_t i;
        for (i=0 ; i<len ; i++)
        {
                array[i] = 0;
        }
}
```

ARR-30. 배열의 인덱스가 유효한 범위 안에 있음을 보장하라.

배열의 참조가 배열의 경계 안에서 일어나게 하는 일은 전적으로 프로그래머의 책임이다.

[문제코드] 아래의 예제에서는 배열 위쪽의 경계만을 보장한다.

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

enum { TABLESIZE=10 };
int *table = NULL;
int insert_in_table(int, int);

int main()
{
        if (insert_in_table(-5, 100) != 0)
        {
                printf("배열에 데이터를 저장하였습니다.\n");
        }
        else
        {
                printf("데이터 저장에 실패하였습니다.");
        }

        getchar();
        return 0;
}

int insert_in_table(int pos, int value)
{
        if (!table)
        {
                table = (int *)malloc(sizeof(int) * TABLESIZE);
```

```
                if (table == NULL)
                {
                        printf("동적메모리할당실패!!!\n");
                        exit(1);
                }
        }

        if (pos >= TABLESIZE)
        {
                return 0;
        }

        table[pos] = value;
        return -1;
}
```

[해결방법]

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

enum { TABLESIZE=10 };
int *table = NULL;
int insert_in_table(size_t, int);

int main()
{
        if (insert_in_table(-5, 100) != 0)
        {
                printf("배열에 데이터를 저장하였습니다.\n");
        }
        else
        {
                printf("데이터 저장에 실패하였습니다.");
        }

        getchar();
        return 0;
}

int insert_in_table(size_t pos, int value)
{
        if (!table)
        {
                table = (int *)malloc(sizeof(int) * TABLESIZE);
                if (table == NULL)
                {
                        printf("동적메모리할당실패!!!\n");
                        exit(1);
                }
        }

        if (pos >= TABLESIZE)
```

```
        {
                return 0;
        }

        table[pos] = value;
        return -1;
}
```

ARR-32. 가변 배열에서 크기를 나타내는 인자가 유효한 범위에 있음을 보장하라.

○ 가변배열은 전통적인 C배열과 근본적으로 동일하다. 주요한 차이점은 선언시 그 크기가 상수표기로 주어지지 않는다는 점이다. 가변배열은 다음과 같이 선언될 수 있다.

```
  char val[s];
```

정수 s와 배열의 선언이 모두 런타임에서 평가된다.

가변배열에 주어지는 크기 값이 정상적이지 않을 경우, 프로그램은 기대하지 않은 방식으로 동작할 수 있다.

[문제코드] 배열의 크기로 사용되어지는 변수 s의 값이 유효한 크기인지 확실치가 않다. s가 음수라면 프로그램 프로그램 스택이 깨지고, 너무 큰 양수이면 스택 오버플로우가 발생할 수 있다.

```
#include <stdio.h>

void func(size_t s)
{
        int val[s];
        val[s-1] = 123;
        printf("val[%d] = %d\n", s-1, val[s-1]);
}

int main()
{
        func(10);

        getchar();
        return 0;
}
```

[해결방법] s가 유효한 범위의 값인지를 확인한다.

```
#include <stdio.h>

enum { MAX_ARRAY = 1024 };

void func(size_t s)
{
        if (s == 0 || s > MAX_ARRAY)
        {
                printf("범위지정오류!!!\n");
                return;
        }

        int val[s];
        val[s-1] = 123;
        printf("val[%d] = %d\n", s-1, val[s-1]);
}

int main()
{
        func(10);

        getchar();
        return 0;
}
```

ARR-33. 충분한 크기의 공간에서 복사가 진행됨을 보장하라.

모든 데이터를 담을 수 있을 만큼 크지 않은 배열에 데이터를 복사하면 버퍼 오버플로우를 발생시킬 수 있다.

[문제코드] memcpy() 함수를 이용하여 src가 가리키는 문자열을 dest에 복사함에 있어서 복사하고자 하는 문자열의 수를 src의 크기로 복사하는 실수를 범하고 있다.

```
#include <stdio.h>

enum { WORKSPACE_SIZE=10 };

void func(const char src[], size_t len)
{
        char dest[WORKSPACE_SIZE];
        memcpy(dest, src, len * sizeof(char));
        /* 복사된 dest를 사용하는 코드가 여기에 */
        printf("복사된 문자열 : %s\n", src);
        printf("복사한 문자열 : %s\n", dest);
}
```

```
int main()
{
        char string[] = "do not edit this string...";
        func(string, sizeof(string)/sizeof(string[0]));

        getchar();
        return 0;
}
```

[해결방법] 복사가 가능한 크기인가를 확인한다.

```
#include <stdio.h>

enum { WORKSPACE_SIZE=10 };

void func(const char src[], size_t len)
{
        char dest[WORKSPACE_SIZE];

        if (len > WORKSPACE_SIZE)
        {
                printf("문자열의 길이가 너무 길어 복사가 불가능합니다.\n");
                return;
        }
        memcpy(dest, src, len * sizeof(char));
        /* 복사된 dest를 사용하는 코드가 여기에 */
        printf("복사된 문자열 : %s\n", src);
        printf("복사한 문자열 : %s\n", dest);
}

int main()
{
        char string[] = "do not edit this string...";
        func(string, sizeof(string)/sizeof(string[0]));

        getchar();
        return 0;
}
```

ARR-34. 표현식에서 배열 타입이 호환 가능함을 보장하라.

표현식에서 호환되지 않는 두 가지 이상의 배열을 사용하려면 정의되지 않은 동작을 초래한다.

[문제코드] 함수 호출 시 넘기는 인자와 인자를 받아 저장하는 파라미터의 타입이 다르다.

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

int getMenu(char (*mStr)[10], size_t mCnt)
{
        int i;
        int menuNum = 0;

        while (menuNum != mCnt)
        {
                printf("작업메뉴를 선택하세요.\n");
                for (i=0 ; i<mCnt ; i++)
                {
                        printf("%d. %s\n", i+1, *(mStr + i));
                }

                __fpurge(stdin);
                printf("메뉴번호입력 : ");
                if (scanf("%d", &menuNum) != 1) continue;
                if (menuNum < 0 || menuNum > mCnt) continue;
                break;
        }
        return menuNum;
}

int main()
{
        int menuNum = 0;
        char *menuStr[] = { "INPUT", "OUTPUT", "EXIT" };
        size_t menuCnt = sizeof(menuStr)/sizeof(menuStr[0]);

        menuNum = getMenu(menuStr, menuCnt);
        printf("%d번을 선택하셨습니다.\n", menuNum);

        getchar();
        return 0;
}
```

[해결방법] 동일한 자료형으로 일치시킨다.

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

int getMenu(char *mStr[], size_t mCnt)
{
        int i;
        int menuNum = 0;

        while (menuNum != mCnt)
        {
                printf("작업메뉴를 선택하세요.\n");
                for (i=0 ; i<mCnt ; i++)
                {
                        printf("%d. %s\n", i+1, *(mStr + i));
                }

                __fpurge(stdin);
                printf("메뉴번호입력 : ");
                if (scanf("%d", &menuNum) != 1) continue;
                if (menuNum < 0 || menuNum > mCnt) continue;
                break;
        }
        return menuNum;
}

int main()
{
        int menuNum = 0;
        char *menuStr[] = { "INPUT", "OUTPUT", "EXIT" };
        size_t menuCnt = sizeof(menuStr)/sizeof(menuStr[0]);

        menuNum = getMenu(menuStr, menuCnt);
        printf("%d번을 선택하셨습니다.\n", menuNum);

        getchar();
        return 0;
}
```

ARR-36. 같은 배열을 참조하고 있지 않다면 두 개의 포인터를 빼거나 비교하지 마라.

ㅇ 두개의 포인터로 뺄셈을 수행하려면 두 포인터가 같은 배열을 참조하거나 적어도 배열의 마지막 원소 다음 부분을 가리켜야 한다.

ㅇ 두 개의 포인터로 뺄셈 연산을 수행하면, 두 원소간의 거리가 계산되는 데, 이때 두원소간의 거리란 두 원소간의 첨자(index) 차이이다.

```
int nums[SIZE];
int *next_num_ptr = nums;
int free_bytes;

/* 배열을 채우면서 next_num_ptr을 증가시킨다. */
free_bytes = (next_num_ptr - nums) * sizeof(int);
printf("free_types : %d\n", free_bytes);
```

제8장 문자열

STR-01. 문자열 관리를 위해 일관된 계획을 사용하여 일관되게 구현하라.

ㅇ 프로그램에서 NULL문자로 종료되는 바이트 문자열을 관리하는데 있어서 두 가지 방법이 있는데, 한 프로젝트 내에서는 둘 중에 한가지 방식을 선택하여 문자열을 일관되게 관리하도록 하라.

1. 문자열을 정적으로 할당된 배열을 통해 관리하는 기법
   초과되는 데이터는 버려지기 때문에 결과값으로 나오는 문자열은 충분히 검증되어야 한다.

2. 요구되는 만큼 메모리를 동적으로 할당하여 사용하는 기법
   입력을 제한하지 않으면 메모리 고갈로 인한 서비스 거부 공격에 사용될 수 있다.

STR-03. NULL문자로 종료된 문자열이 부적절하게 잘리지 않게 하라.

버퍼 오버플로우 취약성을 완화하기 위해 복사되는 바이트의 수를 제한하는 대체 함수들이 제안되곤 하는데 이 함수들은 지정된 제한을 넘는 문자열을 잘라버린다. 이러한 의도하지 않은 잘림은 데이터 손실과 더불어 프로그램의 취약성을 타나내는 원인이 된다.

strncpy(), strncat(), fgets(), snprintf()

[해결방법]

```
char *string_data;
char buff[10];

/* string_data 초기화 */
if (string_data == NULL)
{
        printf("No String\n");
}
else if (strlen(string_data) >= sizeof(buff))
{
        printf("too long\n");
}
else
{
        strncpy(buff, string_data, sizeof(buff));
}
```

STR-04. 기본 문자 집합에서는 문자들을 위해 char형을 사용하라.

ㅇ char형은 signed char 혹은 unsigned char와 동일한 범위, 표기를 갖도록 정의한다. 하지만 둘 중 하나와 같다고는 해도 char형은 이 둘과는 분리된 타입이며, 서로 호환 가능하지도 않다.

ㅇ 표준 문자열 처리 함수들과의 호환성을 위해 문제데이터에는 signed나 unsigned가 아닌 일반 char형을 사용하는 방법이 최선이다.

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

int main()
{
        int len;

        char cstr[] = "char string";
        signed char scstr[] = "signed char string";
        unsigned char ucstr[] = "unsigned char string";

        len = strlen(cstr);
        len = strlen(scstr);    // 경고발생
        len = strlen(ucstr);    // 경고발생

        return 0;
}
```

STR-05. 문자열 상수를 가리키는 포인터는 const로 선언하라.

STR-30. 문자열 리터럴을 수정하려고 하지마라.

문자열 리터럴은 상수이므로 const 지정자에 의해 보호되어야 한다.

[문제코드] 아래의 코드는 런타임 에러를 발생시킨다.

```c
#include <stdio.h>

int main()
{
        char *c = "Hello";
        c[0] = 'C';
        printf("%s\n", c);

        getchar();
        return 0;
}
```

[해결방법] 컴파일 에러 발생

```c
#include <stdio.h>

int main()
{
        const char *c = "Hello";
        c[0] = 'C';
        printf("%s\n", c);

        getchar();
        return 0;
}
```

STR-06. strtok()에서 파싱되는 문자열이 보존된다고 가정하지 마라.

○ strtok()함수는 처음 호출되면 문자열내의 구분자가 처음 나타나는 부분까지 파싱하고, 구분자를 NULL문자로 바꾼 후, 토큰의 처음 주소를 반환한다. 이후, 다시 strtok()함수를 호출하면 가장 최근에 NULL문자로 바뀐 부분부터 파싱이 시작된다.

○ strtok()함수는 인자를 수정하므로 원본 문자열은 안전하지가 않다. 원본 문자열을 보존하고 싶다면 문자열의 복사본을 만들어 사용하도록 하여야 한다.

[문제코드]

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

int main()
{
        char *token, *path;

        path = getenv("PATH");

        printf("파싱 전\n");
        puts(path);
        puts("\n");

        token = strtok(path, ":");
        puts(token);
        while (token = strtok(0, ":"))
        {
                puts(token);
        }
        puts("\n");

        printf("파싱 후\n");
        puts(path);
        puts("\n");

        getchar();
        return 0;
}
```

[해결방법]

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
	char *token, *path, *copy;

	path = getenv("PATH");
	copy = (char *)calloc(strlen(path)+1, sizeof(char));
	if (copy == NULL)
	{
		return 1;
	}
	strncpy(copy, path, strlen(path));

	printf("파싱 전\n");
	puts(path);
	puts("\n");

	token = strtok(copy, ":");
	puts(token);
	while (token = strtok(0, ":"))
	{
		puts(token);
	}
	puts("\n");

	printf("파싱 후\n");
	puts(path);
	puts("\n");

	getchar();
	return 0;
}
```

STR-35. 경계가 불분명한 소스로부터 고정된 길이의 배열에 데이터를 복사하지 마라.

별도의 경계 없이 복사를 수행하는 함수들은 종종 외부 입력에서 적절한 크기가 들어올 것이라고 생각하여 버퍼 오버플로우를 발생시킬 수 있다.

[문제코드] gets()함수는 입력되는 문자열의 길이를 제한할 수 없으므로 버퍼 오버플로우를 발생시킬 수 있다.

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

int main()
{
        char name[10];

        gets(name);
        printf("name : %s\n", name);

        __fpurge(stdin);
        getchar();
        return 0;
}
```

[해결방법] 길이제한이 가능한 fgets()함수를 사용한다.

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

int main()
{
        char name[10];

        fgets(name, sizeof(name), stdin);
        printf("name : %s\n", name);

        __fpurge(stdin);
        getchar();
        return 0;
}
```

[문제코드] scanf() 함수를 이용하여 문자열을 입력 시, 길이를 제한할 수 없다.

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

int main()
{
        char name[10];

        scanf("%s", name);
        printf("%s\n", name);

        __fpurge(stdin);
        getchar();
        return 0;
}
```

[해결방법]

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

#define STRING(n) STRING_AGAIN(n)
#define STRING_AGAIN(n) #n
#define READ_CHARS 9

int main()
{
        char name[READ_CHARS + 1];

        scanf("%"STRING(READ_CHARS)"s", name);
        printf("%s\n", name);

        __fpurge(stdin);
        getchar();
        return 0;
}
```

메모리관리 / 입력과 출력

▶ 메모리 할당 및 해제 시 고려사항
▶ 메모리 할당 에러 처리
▶ 입출력 관련 함수 선언 및 사용시 고려사항

제9장 메모리관리 (MEM)

○ 프로그래머 관점에서의 메모리 관리

- 메모리 할당
- 메모리 해제
- 메모리 읽기
- 메모리 쓰기

▶ 메모리 할당 / 해제 함수

◑ malloc() 함수

기 능	메모리를 동적으로 할당 받는다.	
헤 더	stdlib.h	
선 언	void *malloc(size_t size);	
형식인자	size	필요한 메모리 크기로써 단위는 바이트이다.
리 턴	성공시	할당된 메모리의 시작주소
	실패시	NULL을 반환한다.
설 명	메모리를 동적으로 할당받는 함수로써 인자로 주어진 바이트 수 만큼의 메모리를 할당하고 성공시에는 할당된 메모리의 시작 주소를 반환한다. malloc() 함수에 의해 반환되는 값은 동적으로 할당된 메모리의 시작주소만을 반환한다. 이것은 주소상수만 있을 뿐, 자료형이 정해져 있지 않다는 것이다. 그러므로 malloc()함수를 호출하여 반환받은 포인터를 사용하기 위해서는 그 용도에 알맞도록 캐스팅 하여야 한다. 일반적으로 함수 호출과 동시에 캐스팅을 하여 포인터 변수에 저장한다.	

	※ malloc() 함수에 의해 동적으로 할당된 메모리 공간은 자동으로 초기화가 이뤄지지 않는다. 그러므로 사용하기 전에는 반드시 memset()함수를 이용하여 메모리 영역을 초기화 한 후, 사용하는것이 좋다.

◑ calloc() 함수

기 능	메모리를 동적으로 할당 받는다.	
헤 더	stdlib.h	
선 언	void *malloc(size_t n_elem, size_t size);	
형식인자	n_elem	동적할당 요소의 갯수
	size	요소 하나의 크기
리 턴	성공시	할당된 메모리의 시작주소
	실패시	NULL을 반환한다.
설 명	메모리를 동적으로 할당받는 함수로써 첫번째 인자로 주어진 요소의 갯수 만큼의 공간을 할당받는다. 이때 두번째 인자가 요소 하나의 크기를 나타낸다. 그러므로 n_elem * size 바이트 만큼의 공간이 할당된다. calloc() 함수에 의해 반환되는 값은 동적으로 할당된 메모리의 시작주소만을 반환한다. 이것은 주소상수만 있을 뿐, 자료형이 정해져 있지 않다는 것이다. 그러므로 calloc()함수를 호출하여 반환받은 포인터를 사용하기 위해서는 그 용도에 알맞도록 캐스팅 하여야 한다. 일반적으로 함수 호출과 동시에 캐스팅을 하여 포인터 변수에 저장한다. ※ calloc() 함수에 의해 동적으로 할당된 메모리 공간은 자동으로 초기화가 된다.	

◑ realloc() 함수

기 능	동적으로 할당 받은 메모리 공간의 크기를 변경한다.	
헤 더	stdlib.h	
선 언	void *realloc(void *ptr, size_t size);	
형식인자	ptr	크기를 변경할 메모리주소를 가리키는 포인터
	size	새로 지정할 메모리의 크기
리 턴	성공시	크기를 재조정한 후의 할당된 메모리의 시작주소

	실패시	NULL
설 명	realloc()함수는 동적으로 할당된 메모리공간의 크기를 확장 또는 축소하는 함수이다. 이 함수는 첫번째 인자로 주어진 포인터가 가리키는 메모리 공간의 크기를 두번째 인자로 주어진 크기만큼 확장하거나 축소한다. 이때 축소하는 경우 새로 조정되는 바이트 수 이후의 데이터에 대해서는 버려지게 되며, 확장시에는 확장된 여분의 공간은 초기화가 되지 않은 상태로 확장된다. 만약 realloc()함수에 의해 기존의 메모리 공간을 확장하는데 있어서 확장되는 공간만큼 연속된 공간이 부족한 경우, 새로운 위치에 메모리 공간을 할당받아 기존 메모리 공간의 데이터를 복사한 후, 새로 할당된 메모리 공간의 시작주소를 반환하게 된다. 이때 기존에 사용하던 메모리 공간은 자동으로 해제한다.	

◑ free() 함수

기 능	동적으로 할당받은 메모리 공간을 해제 한다.	
헤 더	stdlib.h	
선 언	void free(void *ptr);	
형식인자	ptr	해제할 메모리공간을 가리키는 포인터
리 턴	없음	
설 명	malloc()함수나 calloc()함수에 의해 동적으로 할당받은 메모리 공간을 해제한다. 메모리 공간을 해제한다는 것은 사용하던 메모리 공간을 수거하여 OS에 반환하므로써 이후 재사용이 가능하도록 하기 위함이다. 만약 동적으로 할당 받아 사용하던 메모리 공간을 해제하지 않는다면 결국 메모리 부족현상으로 인해 시스템이나 프로그램에 심각한 문제를 야기하게 된다. free()함수는 동적으로 할당하여 사용하던 메모리 공간을 해제할 뿐, 해당 공간에 저장되어 있는 데이터를 삭제하지 않는다. 또한 넘겨받은 인자를 NULL로 초기화 하지도 않는다. 그러므로 되도록이면 free()함수를 호출하여 동적으로 할당된 메모리를 해제하기 전에 memset()를 이용하여 메모리상의 데이터를 초기화 한 후 해제하고, 해제하고 난 후에는 해제된 메모리를 가리키는 포인터를 NULL로 초기화 함으로써 해제된 메모리 공간을 참조하는 일이 없도록 하여야 한다.	

▶ 부실한 메모리 관리로 발생 가능한 문제점

- 힙 버퍼 오버플로우
- 댕글링 포인터 (Dangling pointer)
- 중복해제

※ Dangling pointer : 존재하지 않는 대상을 참조하는 포인터

- 초기화 되지 않은 포인터
- free()함수에 의해 해제된 메모리 공간을 가리키는 포인터

MEM-00. 동일한 함수 또는 동일한 파일 내에서 메모리를 할당하고 해제하라.

[문제코드] 각기 다른 함수나 파일에서 메모리를 할당하고 해제하는 것은 중복해제의 취약성이 발생할 수 있다.

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

enum { MIN_SIZE_ALLOWED = 20 };

int verify_list(char *, size_t);
void process_list(size_t);

int main()
{
        process_list(10);

        getchar();
        return 0;
}

int verify_list(char *list, size_t size)
{
        if (size < MIN_SIZE_ALLOWED)
        {
                free(list);
                return -1;
        }
        return 0;
}

void process_list(size_t number)
{
        char *list = (char *)malloc(number);
        if (list == NULL)
        {
                return;
        }
```

```
        if (verify_list(list, number) == -1)
        {
                printf("할당된 메모리의 크기가 %d 미만 입니다.\n",
MIN_SIZE_ALLOWED);
                free(list);
                return;
        }
        /* list 사용 */
        strcpy(list, "memory test");
        printf("%s\n", list);
        free(list);
}
```

[해결방법]

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

enum { MIN_SIZE_ALLOWED = 20 };

int verify_list(char *, size_t);
void process_list(size_t);

int main()
{
        process_list(10);

        getchar();
        return 0;
}

int verify_list(char *list, size_t size)
{
        if (size < MIN_SIZE_ALLOWED)
        {
                //free(list);
                return -1;
        }
        return 0;
}

void process_list(size_t number)
{
        char *list = (char *)malloc(number);
        if (list == NULL)
        {
                return;
        }

        if (verify_list(list, number) == -1)
        {
                printf("할당된 메모리의 크기가 %d 미만 입니다.\n",
```

```
MIN_SIZE_ALLOWED);
                free(list);
                return;
        }
        /* list 사용 */
        strcpy(list, "memory test");
        printf("%s\n", list);
        free(list);
}
```

MEM-01. free()후 즉시 포인터에 새로운 값을 저장하라.

댕글링 포인터는 중복해제나 해제된 메모리에 엑세스 하는 취약성이 있다.

[문제코드]

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdio_ext.h>

enum { STR_SIZE_MAX=20 };
enum { TYPE_1=0, TYPE_2=0 };
char *getMessage();

int main()
{
        char *message;
        size_t message_type;

        message = getMessage();
        if (message == NULL)
        {
                return 1;
        }

        printf("message_type (0 or 1) : ");
        __fpurge(stdin);
        if (scanf("%lu", &message_type) != 1 || message_type > 2)
        {
                printf("입력오류\n");
                return 2;
        }

        if (message_type == TYPE_1)
        {
                /* message를 사용하는 코드 */
                free(message);
        }

        if (message_type == TYPE_2)
        {
```

```
                /* message를 사용하는 코드 */
                free(message);
        }

        __fpurge(stdin);
        getchar();
        return 0;
}

char *getMessage()
{
        char tmp[STR_SIZE_MAX];
        char *nr, *message;

        memset(tmp, '\0', sizeof(tmp));
        printf("메세지 입력 (20바이트 이내) : ");
        __fpurge(stdin);
        if (fgets(tmp, sizeof(tmp), stdin) == NULL)
        {
                return NULL;
        }

        nr = strchr(tmp, '\n');
        *nr = nr ? '\0' : *nr;
        if (tmp[0] == '\0')
        {
                return NULL;
        }

        message = (char *)malloc(strlen(tmp) + 1);
        if (message == NULL)
        {
                return NULL;
        }

        strncpy(message, tmp, strlen(tmp));
        return message;
}
```

[해결방법] 포인터에 할당되었던 메모리를 해제한 후에는 다른 유효한 객체를 참조하게 하거나 NULL값을 지정한다.

※ free()함수는 NULL포인터를 인자로 받는경우, 아무 동작도 하지 않는다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdio_ext.h>

enum { STR_SIZE_MAX=20 };
enum { TYPE_1=0, TYPE_2=0 };
char *getMessage();

int main()
```

```
{
        char *message;
        size_t message_type;

        message = getMessage();
        if (message == NULL)
        {
                return 1;
        }

        printf("message_type (0 or 1) : ");
        __fpurge(stdin);
        if (scanf("%lu", &message_type) != 1 || message_type > 2)
        {
                printf("입력오류\n");
                return 2;
        }

        if (message_type == TYPE_1)
        {
                /* message를 사용하는 코드 */
                free(message);
                message = NULL;
        }

        if (message_type == TYPE_2)
        {
                /* message를 사용하는 코드 */
                free(message);
                message = NULL;
        }

        __fpurge(stdin);
        getchar();
        return 0;
}

char *getMessage()
{
        char tmp[STR_SIZE_MAX];
        char *nr, *message;

        memset(tmp, '\0', sizeof(tmp));
        printf("메세지 입력 (20바이트 이내) : ");
        __fpurge(stdin);
        if (fgets(tmp, sizeof(tmp), stdin) == NULL)
        {
                return NULL;
        }

        nr = strchr(tmp, '\n');
        *nr = nr ? '\0' : *nr;
        if (tmp[0] == '\0')
        {
                return NULL;
```

```
        }

        message = (char *)malloc(strlen(tmp) + 1);
        if (message == NULL)
        {
                return NULL;
        }

        strncpy(message, tmp, strlen(tmp));
        return message;
}
```

☞ 예외) 함수나 block내의 auto변수는 함수가 종료되거나 블럭을 탈출하면서 자동으로 소멸되므로 NULL값을 지정할 필요가 없다.

MEM-02. 메모리 할당 함수의 반환값을 즉시 할당된 타입의 포인터로 변환시켜라.

[문제코드] 메모리 할당 함수의 반환값은 void *형이므로 변환없이 사용할 경우, 실제 할당된 메모리와 다른 크기의 메모리를 사용하게 되는 경우가 발생한다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdio_ext.h>

typedef struct _gadget
{
        int i;
        double d;
} gadget;

typedef struct _widget
{
        char c[8];
        int i;
        double d;
} widget;

int main()
{
        widget *p;
        p = malloc(sizeof(gadget));
        if (p == NULL)
        {
                return 1;
        }

        p->i = 0;
        p->d = 0.0;
        strcpy(p->c, "apple");
        printf("p->c : %s\n", p->c);
        printf("p->i : %d\n", p->i);
        printf("p->d : %.2lf\n", p->d);
```

```
        free(p);
        getchar();
        return 0;
}
```

[해결방법]

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdio_ext.h>

typedef struct _gadget
{
        int i;
        double d;
} gadget;

typedef struct _widget
{
        char c[8];
        int i;
        double d;
} widget;

int main()
{
        widget *p;

        /*
                잘못된 주소 타입 저장으로 경고 발생
                p = (gadget *)malloc(sizeof(gadget));
                바른 할당과 바른 주소 저장으로 오류 없음
                p = (widget *)malloc(sizeof(widget));
        */
        p = (widget *)malloc(sizeof(widget));
        if (p == NULL)
        {
                return 1;
        }

        p->i = 0;
        p->d = 0.0;
        strcpy(p->c, "apple");
        printf("p->c : %s\n", p->c);
        printf("p->i : %d\n", p->i);
        printf("p->d : %.2lf\n", p->d);

        free(p);
        getchar();
        return 0;
}
```

▶ 자주 사용할 메모리 동적할당 메크로

```
#define MALLOC(number, type) ((type *)malloc((number) * sizeof(type)))

#define CALLOC(number, type) ((type *)calloc((number), sizeof(type)))

#define REALLOC(pointer, number, type) \
        ((type *)realloc(pointer, (number), sizeof(type)))

#define ALLOC_ERROR_CHECK(pointer) \
        if (pointer == NULL) { printf("메모리할당에러\n"); exit; }
```

MEM-03. 재사용을 위해 반환하는 리소스에 있는 중요한 정보를 초기화 하라.

[문제코드] 재사용하게 된 리소스에 민감한 데이터가 들어가 있는 경우, 사용권한이 없는 사용자에게 해당 데이터가 노출될 수 있다.

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

#define MALLOC(number, type) \
        ((type *)malloc((number) * sizeof(type)))
#define ALLOC_ERROR_CHECK(pointer) \
        if (pointer == NULL) \
        { \
                printf("메모리할당오류\n"); \
                exit(1); \
        }

int main()
{
        char *secret;
        char *new_secret;
        size_t size;

        secret = MALLOC(15, char);
        ALLOC_ERROR_CHECK(secret);
        strcpy(secret, "800215-1194919");
        printf("secret의 내용 : %s\n", secret);

        size = strlen(secret);
        new_secret = MALLOC(size+1, char);
        ALLOC_ERROR_CHECK(new_secret);
        strcpy(new_secret, secret);
        printf("new_secret의 내용 : %s\n", new_secret);

        free(new_secret);
```

```
        new_secret = NULL;
        free(secret);
        secret = NULL;

        getchar();
        return 0;
}
```

[해결방법]

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

#define MALLOC(number, type) \
        ((type *)malloc((number) * sizeof(type)))
#define ALLOC_ERROR_CHECK(pointer) \
        if (pointer == NULL) \
        { \
                printf("메모리할당오류\n"); \
                exit(1); \
        }

int main()
{
        char *secret;
        char *new_secret;
        size_t size;

        secret = MALLOC(15, char);
        ALLOC_ERROR_CHECK(secret);
        strcpy(secret, "800215-1194919");
        printf("secret의 내용 : %s\n", secret);

        size = strlen(secret);
        new_secret = MALLOC(size+1, char);
        ALLOC_ERROR_CHECK(new_secret);
        strcpy(new_secret, secret);
        printf("new_secret의 내용 : %s\n", new_secret);

        memset(new_secret, '\0', size);
        free(new_secret);
        new_secret = NULL;
        memset(secret, '\0', size);
        free(secret);
        secret = NULL;

        getchar();
        return 0;
}
```

realloc()함수를 이용하여 할당된 메모리 공간을 확장 시, 만약 현재 할당된 공간뒤에 확장되는 공간만큼의 여분의 공간이 없는경우, realloc()함수는 새로운 위치에 공간을 할당하여 기존의 데이터를 복사하고, 기존의 메모리공간을 해제한 후, 새로 할당된 메모리의 시작주소를 반환한다.

이때 기존에 사용하던 메모리 공간을 해제하지만 데이터 초기화까지 실행되지는 않으므로 중요한 데이터가 그대로 남은 채, 다른 용도로 할당될 수 도 있다.

[문제코드]

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

#define MALLOC(number, type) \
        ((type *)malloc((number) * sizeof(type)))
#define REALLOC(pointer, number, type) \
        ((type *)realloc(pointer, (number) * sizeof(type)))
#define ALLOC_ERROR_CHECK(pointer) \
        if (pointer == NULL) \
        { \
                printf("메모리할당오류\n"); \
                exit(1); \
        }

int main()
{
        char *secret;
        size_t size;

        secret = MALLOC(7, char);
        ALLOC_ERROR_CHECK(secret);
        strcpy(secret, "800215");
        printf("secret의 내용 : %s\n", secret);

        size = strlen(secret);
        REALLOC(secret, size+8+1, char);
        ALLOC_ERROR_CHECK(secret);
        strcat(secret, "-1194919");
        printf("secret의 내용 : %s\n", secret);

        size = strlen(secret);
        memset(secret, '\0', size);
        free(secret);
        secret = NULL;

        getchar();
        return 0;
}
```

[해결방법] realloc()함수에 의존하지 않고 realloc()함수와 비슷하게 동작하는 로직을 만들어서 해결하라.

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

#define MALLOC(number, type) \
        ((type *)malloc((number) * sizeof(type)))
#define REALLOC(pointer, number, type) \
        ((type *)realloc(pointer, (number) * sizeof(type)))
#define ALLOC_ERROR_CHECK(pointer) \
        if (pointer == NULL) \
        { \
                printf("메모리할당오류\n"); \
                exit(1); \
        }

int main()
{
        char *secret;
        char *tmp_buff;
        size_t size;

        secret = MALLOC(7, char);
        ALLOC_ERROR_CHECK(secret);
        strcpy(secret, "800215");
        printf("secret의 내용 : %s\n", secret);

        size = strlen(secret);
        tmp_buff = MALLOC(size+8+1, char);
        ALLOC_ERROR_CHECK(tmp_buff);
        memcpy(tmp_buff, secret, size+1);
        strcat(tmp_buff, "-1194919");

        memset(secret, '\0', size);
        free(secret);
        secret = tmp_buff;
        tmp_buff = NULL;
        printf("secret의 내용 : %s\n", secret);

        getchar();
        return 0;
}
```

MEM-05. 큰 스택 할당을 피하라.

ㅇ 문제점

특히 스택의 증가가 공격에 의해 저어되거나 영향 받을 수 있는 경우라면 더욱 피해야 한다.

ㅇ 해결방법

가변배열을 사용하기 보다 메모리를 동적으로 할당하여 사용한다.

ㅇ 문제점

재귀 함수 역시 스택 할당 초과를 초래할 수 있다. 재귀 함수들은 과도한 재귀 호출 동작으로 인해 스택을 고갈시켜버리지 않는지를 반드시 보장하여야 한다.

ㅇ 해결방법

재귀 함수를 사용하지 않고, 반복문을 이용하여 스택 할당 초과 문제를 해결한다.

MEM-06. 중요한 데이터가 디스크에 기록되지 않도록 보장하라.

ㅇ 개발자는 비밀번호, 암호키 혹은 여타 중요한 정보가 부적절하게 노출되지 않도록 보호해야 하는데 그 중에는 데이터가 디스크에 기록되지 않도록 해야 한다.

ㅇ 데이터가 부적절하게 디스크에 기록되는 두 가지 메커니즘

① Swapping

가상메모리 관리를 구현한 범용 운영체제의 메인 메모리와 디스크와 같은 외부 저장장치 사이에 페이지 단위로 데이터를 이동시키는 paging기법

② Core dump

나중에 디버거로 조사하기 위해 프로세스 메모리의 상태를 디스크에 기록한 파일이다. Core dump는 일반적으로 프로그램이 비정상적으로 종료되거나 에러로 인해 프로그램이 잘못되는 경우, 혹은 프로그램의 종료를 요구하는 시그널을 받을 때 발생한다.

ㅇ 해결방법

리소스 제한을 조절하는 POSIX 표준 시스템콜 setrlimit()를 이용해서 core dump의 크기를 0으로 설정하여 core dump에 의한 중요 정보가 디스크에 기록되는 것을 방지한다.

또한 mlock()을 사용한 메모리 잠금을 통해 페이징이 되는 것을 막을 수 있다. (Windows 시스템에서는 VirtualLoc() 사용)

※ POSIX

서로 다른 UNIX OS의 공통적인 API를 정리하여 이식성이 높은 유닉스 응용프로그램을 개발하기 위한 목적으로 IEEE가 책정한 어플리케이션 인터페이스 규격이다.

유닉스 운영체제에 기반을 두고 있는 일련의 표준 운영체제 인터페이스이다.

※ 보안과 관련된 어플리케이션에서 중요한 데이터는 가능하다면 암호화 하도록 하여야 한다.

MEM-07. calloc()의 인자가 곱해지는 경우, size_t로 표현될 수 있게 하라.

calloc()하수는 두 개의 인자(할당할 원소의 개수, 원소의 크기)를 곱한 크기로 메모리를 할당하는데 이때 할당된 크기가 size_t로 표현 가능한 인자인가를 확인해야 한다.

[문제점] calloc()함수로 할당된 메모리의 크기가 요구된 크기보다 작은 크기의 메모리가 할당되면 이 상태에서 버퍼에 데이터가 복사되면 오버플로우가 발생한다.

[해결방법] calloc()함수 호출전에 할당될 메모리의 크기가 size_t 타입에 저장할 수 있는 크기인지 확인한 후 호출한다.

MEM-08. 동적으로 할당된 배열을 resize 하는 경우에만 realloc()함수를 사용하라.

realloc()으로 재할당되는 배열은 동일한 타입이어야 하는데 다른 타입의 기억공간으로 재할당하면 realloc()함수에의해 복사된 데이터들이 제대로 사용될 수 없다.

[문제코드]

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

enum { ALLOC_LEN=2 };

int main()
{
        int *iptr;
        short *sptr;
        int i;

        iptr = (int *)malloc(ALLOC_LEN * sizeof(int));
        if (iptr == NULL)
        {
                return 1;
        }
        *(iptr + 0) = 0xAAAABBBB;
        *(iptr + 1) = 0xCCCCDDDD;

        sptr = (short *)realloc(iptr, ALLOC_LEN * 2 * sizeof(int));
        if (sptr == NULL)
        {
                return 1;
        }

        printf("sptr[0] : %hX\n", *(sptr+0));
        printf("sptr[1] : %hX\n", *(sptr+1));
        printf("sptr[2] : %hX\n", *(sptr+2));
        printf("sptr[3] : %hX\n", *(sptr+3));

        getchar();
        return 0;
```

```
}
```

[해결방법]

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

enum { ALLOC_LEN=2 };

int main()
{
        int *iptr;
        int *sptr;
        int i;

        iptr = (int *)malloc(ALLOC_LEN * sizeof(int));
        if (iptr == NULL)
        {
                return 1;
        }
        *(iptr + 0) = 0xAAAABBBB;
        *(iptr + 1) = 0xCCCCDDDD;

        sptr = (int *)realloc(iptr, ALLOC_LEN * 2 * sizeof(int));
        if (sptr == NULL)
        {
                return 1;
        }

        printf("sptr[0] : %X\n", *(sptr+0));
        printf("sptr[1] : %X\n", *(sptr+1));
        printf("sptr[2] : %X\n", *(sptr+2));
        printf("sptr[3] : %X\n", *(sptr+3));

        getchar();
        return 0;
}
```

MEM-09. 메모리 할당 루틴이 메모리를 초기화 해줄 것이라고 가정하지 마라.

malloc()함수로 할당된 메모리나, realloc()함수로 할당된 메모리 중 추가된 메모리의 경우, 초기화 되어 있지 않거나 다른 섹션(혹은 다른 프로그램)에서 사용하던 데이터를 가지고 있을 수 도 있다.

[문제코드] malloc()함수로 할당된 메모리에 strncpy()함수를 이용하여 문자열을 복사하면 문자열의 끝에 NULL문자가 자동으로 저장되지 않는다.

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

enum { MAX_BUF_SIZE = 256 };

int main()
{
        char *str = "dream & hope";
        char *buf;
        size_t len;

        len = strlen(str);
        if (len == 0 || len >= MAX_BUF_SIZE - 1)
        {
                printf("문자열의 길이가 0이거나 너무 길어서 처리할 수
없습니다.\n");
                return 1;
        }

        buf = (char *)malloc(MAX_BUF_SIZE * sizeof(char));
        if (buf == NULL)
        {
                return 1;
        }

        /* strncpy() 함수는 지정된 바이트수 만큼 복사한 후,
           NULL을 추가하지 않는다. */
        strncpy(buf, str, len);
        printf("buf : %s\n", buf);
        free(buf);
        getchar();
        return 0;
}
```

[해결방법 1] 문자열을 복사한 후, 명시적으로 NULL문자를 추가한다.

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

enum { MAX_BUF_SIZE = 256 };

int main()
{
        char *str = "dream & hope";
        char *buf;
        size_t len;

        len = strlen(str);
        if (len == 0 || len >= MAX_BUF_SIZE - 1)
        {
                printf("문자열의 길이가 0이거나 너무 길어서 처리할 수
없습니다.\n");
                return 1;
        }

        buf = (char *)malloc(MAX_BUF_SIZE * sizeof(char));
        if (buf == NULL)
        {
                return 1;
        }

        strncpy(buf, str, len);
        /* 문자열을 복사한 후, NULL을 추가 */
        *(buf + len) = '\0';
        printf("buf : %s\n", buf);
        free(buf);
        getchar();
        return 0;
}
```

[해결방법 2] malloc()함수에 의해 할당받은 메모리 공간을 사용하기전에 NULL로 초기화 한다.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

enum { MAX_BUF_SIZE = 256 };

int main()
{
	char *str = "dream & hope";
	char *buf;
	size_t len;

	len = strlen(str);
	if (len == 0 || len >= MAX_BUF_SIZE - 1)
	{
		printf("문자열의 길이가 0이거나 너무 길어서 처리할 수 없습니다.\n");
		return 1;
	}

	buf = (char *)malloc(MAX_BUF_SIZE * sizeof(char));
	if (buf == NULL)
	{
		return 1;
	}

	/* malloc()함수에 의해 할당된 공간을 NULL로 초기화 */
	memset(buf, '\0', MAX_BUF_SIZE);
	strncpy(buf, str, len);
	printf("buf : %s\n", buf);
	free(buf);
	getchar();
	return 0;
}
```

MEM-30. 해제된 메모리에 접근하지 마라.

이미 해제된 메모리에 접근하면 heap상의 데이터 구조가 손상될 수 있다.

[문제코드]

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

typedef struct _node Node;
typedef struct _list List;

struct _node
{
        Node *next;
        char name[20];
        int age;
};

struct _list
{
        Node *head;
};

int add_node(List *, char *, int);
void print_node(List *);
void destroy(List *);

int main()
{
        List *list;
        list = (List *)calloc(1, sizeof(List));
        if (list == NULL)
        {
                return 1;
        }
        list->head = (Node *)calloc(1, sizeof(Node));
        if (list->head == NULL)
        {
                return 1;
        }

        add_node(list, "김기희", 20);
        add_node(list, "홍길동", 19);
        add_node(list, "심청이", 16);
        print_node(list);


        destroy(list);
        getchar();
        return 0;
}
```

```
int add_node(List *list, char *name, int age)
{
        Node *tmp;
        Node *node;
        node = (Node *)calloc(1, sizeof(Node));
        if (node == NULL)
        {
                printf("노드생성실패!\n");
                return 0;
        }
        strcpy(node->name, name);
        node->age = age;

        for (tmp = list->head ; tmp->next != NULL ; tmp = tmp->next);
        tmp->next = node;

        return 1;
}

void print_node(List *list)
{
        Node *tmp;

        for (tmp = list->head->next ; tmp != NULL ; tmp = tmp->next)
        {
                printf("이름 : %s\n", tmp->name);
                printf("연령 : %d\n", tmp->age);
        }
}

void destroy(List *list)
{
        Node *p;
        for (p = list->head->next ; p != NULL ; p = p->next)
        {
                free(p);
        }
}
```

[해결방법]

```
void destroy(List *list)
{
        Node *p, *tmp;

        for (p = list->head->next ; p != NULL ; p = tmp)
        {
                tmp = p->next;
                free(p);
        }
}
```

MEM-31. 동적으로 할당된 메모리는 한 번만 해제하라.

ㅇ 메모리를 여러번 해제하면 해제된 메모리에 접근하는것과 같은 결과를 초래한다. (double free)

ㅇ 힙 상의 데이터 구조가 손상되고 프로그램의 보안에 취약성을 유발한다.

ㅇ double free의 취약성을 제거하려면 동적 메모리가 정확히 한 번만 해제되도록 보장해야 한다.

MEM-32. 메모리 할당 에러를 찾아 해결하라.

ㅇ 동적메모리 할당함수는 요청된 메모리 할당이 실패하면 NULL포인터를 반환한다.
ㅇ 메모리 관리 에러를 발견하고 적절히 처리하지 않으면 프로그램이 예측할 수 없는 동작을 일으킬 수 있다.
ㅇ 따라서 메모리 관리 루틴의 최종 상태를 체크하여 적절히 에러를 처리해 주어야 한다.

MEM-33. 유연한 배열 원소에 정확한 문법을 사용하라.

ㅇ 유연한 배열 맴버(flexible array member)란 구조체가 이름이 있는 한 개 이상의 맴버를 가질 때, 마지막 원소가 불완전한 배열 타입의 맴버를 갖는 경우를 말한다.

즉 배열의 크기가 구조체 내에서 명시적으로 지정되지 않은 경우이다.

ㅇ 유연한 배열 맴버는 다음과 같은 특징을 갖는다.

1) 유연한 배열 맴버는 반드시 구조체의 마지막 원소여야 한다.
2) 유연한 배열 맴버를 원소로 갖는 구조체는 배열로 선언하여 사용할 수 없다.
3) 유연한 배열 맴버를 포함하는 구조체를 다른 구조체의 맴버를 사용할 때에는 마지막 맴버이어야만 한다.
4) sizeof연산자는 유연한 배열 맴버에 적용할 수 없다.
(단, 구조체의 크기는 sizeof 연산자로 구할 수 있으며, 유연한 배열 맴버의 크기는 0으로 계산된다.)

▶ 유연한 배열 맴버(flexible array member)를 갖는 구조체 매크로의 예

```
#define MALLOC_FLEX(stype, number, etype) \
    ((stype *)malloc(sizeof(stype) + (number) * sizeof(etype)))
#define REALLOC_FLEX(pointer, stype, number, etype) \
    ((stype *)realloc(pointer, sizeof(stype) + (number) * sizeof(etype)))
```

[예제] 유연한 배열맴버의 사용예

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MALLOC_FLEX(stype, number, etype) \
   ((stype *) malloc(sizeof(stype) + (number) * sizeof(etype)))
#define REALLOC_FLEX(pointer, stype, number, etype) \
   ((stype *) realloc(pointer, sizeof(stype) + (number) * sizeof(etype)))

typedef struct string
{
        size_t length;
        /* 유연한 배열맴버는 오직 하나만 사용가능하며,
           마지막 맴버여야만 한다. */
        char  text[];
} string;

int main()
{
        string *fruit_name;

        fruit_name = MALLOC_FLEX(string, 7, char);
        if(!fruit_name)
        {
                printf("메모리 할당 에러\n");
                exit(0);
        }
        strcpy( fruit_name ->text, "banana");
        fruit_name->length = strlen(fruit_name->text);
        printf("%s : %lu Byte\n", fruit_name->text, fruit_name->length);

        fruit_name = REALLOC_FLEX(fruit_name, string, 11, char);
        if(!fruit_name)
        {
                printf("메모리 할당 에러\n");
                exit(0);
        }
        strcpy( fruit_name ->text, "pine apple");
        fruit_name->length = strlen(fruit_name->text);
        printf("%s : %lu Byte\n", fruit_name->text, fruit_name->length);
        free(fruit_name);

        getchar();
        return 0;
}
```

유연한 배열 맴버를 구조체의 마지막 맴버로 배치할 때, 배열의 크기를 1로 지정하면 C99에서는 배열의 원소가 1개인 배열로 인식한다. 이때 유연한 배열 맴버의 원소로 n개의 메모리를 동적할당을 하여도 0번 원소만이 존재하게 된다.

[문제코드]

```c
#include <stdio.h>
typedef struct string
{
        size_t length;
        char  text[1];
} string;

int main()
{
        printf("sizeof(string) : %d\n", sizeof(string));

        getchar();
        return 0;
}
```

[해결방법]

```c
#include <stdio.h>
typedef struct string
{
        size_t length;
        char  text[];
} string;

int main()
{
        printf("sizeof(string) : %d\n", sizeof(string));

        getchar();
        return 0;
}
```

제10장 입력과 출력 (FIO)

FIO-00. 포맷문자열을 사용할 때 주의하라.

○ 포맷문자열을 만들 때 실수하기 쉬운 부분

1) 유효하지 않은 변환 지정자(conversion specifiers)의 사용
(변환지정자 : d, i, o, u, x, f, e, g, c, s ......)

2) 부정확한 지정자에 대한 길이 수정자(length modifier)의 사용
(길이수정자 : field 폭)

3) 인자와 변환 지정자의 타입 불일치
(짝이 되는 변환지정자와 타입 : %d - int, %f - float, %lf - double)

4) 유효하지 않은 문자 클래스의 사용
[a-z] a부터 z까지의 문자를 받아들임
[^a-z] a부터 z까지의 문자를 제외한 문자를 받아들임

[예제] sscanf() 함수의 사용예

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

int main()
{
        char a[10] = {};
        char b[10] = {};
        char c[10] = {};
        char op[10] = {};
        char tmp[10] = {};
        char *buf1 = "123+456";
        char *buf2 = "mylovec@gmail.com";

        sscanf(buf1, "%[0-9]%[^0-9]%s", a, op, b);
        printf("a : %s\t op : %s\t b : %s\n", a, op, b);

        memset(a, '\0', sizeof(a));
        memset(b, '\0', sizeof(b));

        sscanf(buf2, "%[^.@]%[.@]%[^.@]%[.@]%s", a, tmp, b, tmp, c);
        printf("a : %s\t b : %s\t c : %s\n", a, b, c);

        getchar();
        return 0;
}
```

FIO-01. 파일이름이나 식별자를 사용하는 함수를 사용할 때에는 주의하라.

ㅇ 파일의 이름만으로 실제 파일에 대한 정보를 얻을 수 없다.

파일객체와 파일 이름의 바인딩은 파일 이름이 사용되는 연산이 있을 때에만 확인된다.

ㅇ 파일명으로 파일을 식별하는 함수

remove(), rename(), fopen(), freopen(), ...

ㅇ 파일 디스크립터와 FILE 포인터는 운영체제에 의해 실제 파일객체와 바인딩 되어 있어 파일명보다는 더 정확하게 파일 객체를 파악할 수 있다.

FIO-03. fopen()함수나 open()함수의 파일 생성에 대하여 특정 조건을 가정하지 마라.

ㅇ fopen()함수나 open()함수는 존재하는 파일을 열거나 파일을 새로 생성할 때 사용한다. 이때 프로그래머의 의도와는 달리 기존의 파일을 덮어쓸 가능성이 있다.

[문제코드] fopen()함수의 w모드는 기존파일의 데이터를 삭제한다.

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

int main()
{
        const char *file = "data.txt";
        /* 기존의 파일 데이터를 삭제한다 */
        FILE *fp = fopen(file, "wb");
        if (fp == NULL)
        {
                printf("파일오픈실패!\n");
                return 1;
        }
        /* fp를 사용하는 코드 */
        fclose(fp);
        getchar();
        return 0;
}
```

[해결방법]

```c
#include <stdio.h>

int main()
{
	const char *file = "data.txt";
	/* 동일한 이름의 파일이 있으면 파일 생성 실패 */
	FILE *fp = fopen(file, "wbx");
	if (fp == NULL)
	{
		printf("파일오픈실패!\n");
		return 1;
	}
	/* fp를 사용하는 코드 */
	fclose(fp);

	getchar();
	return 0;
}
```

[문제코드] 파일에 데이터를 기록하면 기존 데이터가 삭제된다.

```c
#include <stdio.h>
#include <string.h>
/* open() */
#include <fcntl.h>
/* write(), read(), close() */
#include <unistd.h>

int main()
{
	int fd;
	char buffer[100];

	const char *file = "data.txt";
	fd = open(file, O_CREAT | O_WRONLY, 755);
	if (fd == -1)
	{
		printf("파일열기실패!\n");
		return 1;
	}

	printf("저장할 문자열 : ");
	if (fgets(buffer, sizeof(buffer), stdin) != NULL)
	{
		write(fd, buffer, strlen(buffer));
	}

	close(fd);
```

```
        getchar();
        return 0;
}
```

[해결방법]

```
#include <stdio.h>
#include <string.h>
/* open() */
#include <fcntl.h>
/* write(), read(), close() */
#include <unistd.h>

int main()
{
        int fd;
        char buffer[100];

        const char *file = "data.txt";
        fd = open(file, O_CREAT | O_WRONLY | O_EXCL , 755);
        if (fd == -1)
        {
                printf("파일열기실패!\n");
                return 1;
        }

        printf("저장할 문자열 : ");
        if (fgets(buffer, sizeof(buffer), stdin) != NULL)
        {
                write(fd, buffer, strlen(buffer));
        }

        close(fd);

        getchar();
        return 0;
}
```

FIO-05. 여러 파일 속성을 통해 파일을 식별하라.

open해서 사용했던 파일과 새로 open해서 사용하는 파일에 대해 속성을 비교하려면 파일명만 비교하는 것보다, 더 확실하게 같은 파일임을 보장할 수 있다.

fstat() 함수는 파일의 속성을 struct stat구조제체 저장하여 반환한다. 이때 stat구조체의 맴버를 이용하여 파일을 비교할 수 있다.

ㅇ struct stat 구조체의 중요맴버

- st_ino : 파일의 inode값 저장
- st_dev : 파일이 저장된 블럭장치번호

[예제] fstat() 함수를 이용한 파일검증

```
#include <stdio.h>
#include <sys/stat.h>
#include <io.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main()
{
        char *file = "data.txt";
        int fd;
        int len;
        char str[] = "Oh! Happy Day!";
        char buf[100];
        struct stat st_org;
        struct stat st_new;

        fd = open(file, O_CREAT | O_WRONLY, 755);
        if (fd == -1)
        {
                printf("파일열기실패(1)!\n");
                return 1;
        }

        write(fd, str, strlen(str));
        if (fstat(fd, &st_org) == -1)
        {
                printf("파일정보획득실패(1)!\n");
                return 2;
        }
        close(fd);

        fd = open(file, O_RDONLY);
        if (fd == -1)
        {
                printf("파일열기실패(2)!!!");
```

```
                return 1;
        }

        if (fstat(fd, &st_new) == -1)
        {
                printf("파일정보획득실패(2)\n");
                return 2;
        }

        if (st_org.st_ino != st_new.st_ino
                        || st_org.st_dev != st_new.st_dev)
        {
                printf("이전 파일과 다른 파일입니다.\n");

        }
        else
        {
                len = read(fd, buf, sizeof(buf));
                str[len] = '\0';
                printf("readline : %s\n", buf);
        }
        close(fd);

        getchar();
        return 0;
}
```

[예제] 한번의 개방으로 읽기와 쓰기

```
#include <stdio.h>

int main()
{
        size_t len;
        char *file = "data.txt";
        FILE *fp;

        char str[] = "Oh! Happy Day!";
        char buf[100];

        fp = fopen(file, "w+b");
        if (fp == NULL)
        {
                printf("파일열기실패\n");
                return 1;
        }
        len = fwrite(str, sizeof(str[0]), sizeof(str)/sizeof(str[0]), fp);
        printf("%u개의 문자를 기록하였습니다.\n", len);
        printf("기록한 문자열 : %s\n", str);

        fseek(fp, 0L, SEEK_SET);
        len = fread(buf, sizeof(buf[0]), sizeof(buf)/sizeof(buf[0]), fp);
```

```
        printf("%u개의 데이터를 읽어들였습니다.\n", len);
        printf("읽어들인 문자열 : %s\n", buf);

        fclose(fp);
        getchar();
        return 0;
}
```

FIO-07. rewind()보다 fseek()를 사용하라.

파일 위치 표시자를 파일의 처음으로 옮기고자 하는 경우네는 rewind() 함수보다는 fseek() 함수를 사용하라.

rewind()함수는 함수실행의 성공여부를 확인할 수 없다.

[문제코드]

```
#include <stdio.h>

int main()
{
        size_t len;
        char *file = "data.txt";
        FILE *fp;

        char str[] = "Oh! Happy Day!";
        char buf[100];

        fp = fopen(file, "w+b");
        if (fp == NULL)
        {
                printf("파일열기실패\n");
                return 1;
        }
        len = fwrite(str, sizeof(str[0]), sizeof(str)/sizeof(str[0]), fp);
        printf("%u개의 문자를 기록하였습니다.\n", len);
        printf("기록한 문자열 : %s\n", str);

        /* 커서제어의 성공여부 확인 불가 */
        rewind(fp);
        len = fread(buf, sizeof(buf[0]), sizeof(buf)/sizeof(buf[0]), fp);
        printf("%u개의 데이터를 읽어들였습니다.\n", len);
        printf("읽어들인 문자열 : %s\n", buf);

        fclose(fp);
        getchar();
        return 0;
}
```

[해결방법]

```
#include <stdio.h>

int main()
{
        size_t len;
        char *file = "data.txt";
        FILE *fp;

        char str[] = "Oh! Happy Day!";
        char buf[100];

        fp = fopen(file, "w+b");
        if (fp == NULL)
        {
                printf("파일열기실패\n");
                return 1;
        }
        len = fwrite(str, sizeof(str[0]), sizeof(str)/sizeof(str[0]), fp);
        printf("%u개의 문자를 기록하였습니다.\n", len);
        printf("기록한 문자열 : %s\n", str);

        if (fseek(fp, 0L, SEEK_SET) == -1)
        {
                printf("커서이동실패\n");
        }
        else
        {
                len = fread(buf, sizeof(buf[0]),
sizeof(buf)/sizeof(buf[0]), fp);
                printf("%u개의 데이터를 읽어들였습니다.\n", len);
                printf("읽어들인 문자열 : %s\n", buf);
        }

        fclose(fp);
        getchar();
        return 0;
}
```

FIO-08. 열린 파일에 대해 remove()를 호출하면 이미 지워진 파일에 읽기/쓰기 작업을 수행하다가 비정상적으로 프로그램이 종료될 수 있으며, 파일이 의도대로 삭제되지 않아 예상하지 못한 정보유출을 초래할 수 있다.

○ 엄격한 제거를 위해서는 unlink()함수를 이용한다.

unlink() : 파일과 파일 시스템 간의 연결을 끊지만 이 파일에 대한 모든 참조가 닫힐 때 까지 파일을 디스크에 보관한다.

호환성을 고려한다면 열린파일에 대하여 remove()함수를 호출하면 안된다.
(Windows환경에서는 열린파일에 대한 remove()함수와 unlink()함수의 호출은 모두 실패한다.)

```
int access(const char *path, int mode);
```

access()함수는 권한이 있거나 파일이 존재하는 경우 0, 권한이 없거나 파일이 존재하지 않는 경우 -1을 반환한다.

모드	설명
F_OK	파일 및 디렉토리가 존재하는가를 검사
W_OK	쓰기 접근이 허용되는가를 검사
R_OK	읽기 접근이 허용되는가를 검사
X_OK	실행이 허용되는가를 검사

[예제] unlink()함수를 이용한 파일삭제

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

int main()
{
        char *file = "data.txt";

        if (access(file, 2) == 0)
        {
                unlink(file);
        }

        getchar();
        return 0;
}
```

FIO-09. 시스템간에 바이너리 데이터를 전송할 때는 주의하라.

ㅇ 시스템이나 플랫폼마다 구조체 정렬, 부동소수점 모델, 엔디안 등의 속성값이 각기 다르므로 바이너리 데이터의 호환이 불가능 할 수 도 있다.

아래의 예제에서 구조체 Sawon은 시스템마다 크기가 다를 수 있다.

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

typedef struct _sawon
{
        char name[24];
        int grade;
        long pay;

} Sawon;

int main()
{
        int len;
        const char *file = "data.dat";
        FILE *fp;
        Sawon sa[2] = {
                                {"홍길동", 1, 2000000},
                                {"심청이", 2, 1800000}
                };

        printf("%d\n", sizeof(Sawon));
        fp = fopen(file, "wb");
        if (fp == NULL)
        {
                printf("파일열기실패!\n");
                return 1;
        }

        len = fwrite(sa, sizeof(sa[0]), sizeof(sa)/sizeof(sa[0]), fp);
        if (len == -1)
        {
                printf("데이터쓰기실패\n");
        }
        else
        {
                printf("%d바이트를 기록하였습니다.\n", len);
        }

        fclose(fp);

        getchar();
        return 0;
}
```

FIO-10. rename()함수를 사용할 때에는 주의하라.

[문제점] rename()에서 지정된 새 파일명에 해당하는 파일이 기존에 존재하는 경우, 시스템에 따라 다르게 동작한다.

- ㅇ POSIX : 이미 존재하는 대상 파일을 제거
- ㅇ Windows : rename() 실패

[해결방법]

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>

int main()
{
        const char *dest_file = "data.dat~";
        const char *src_file = "data.dat";

        if (access(dest_file, F_OK) == -1)
        {
                if (rename(src_file, dest_file) < 0)
                {
                        printf("이름변경실패\n");
                }
        }
        else
        {
                printf("동일한 이름의 파일이 존재합니다.\n");
        }

        getchar();
        return 0;
}
```

입력과 출력(FIO)

존재하는 파일 제거하고 rename하기 (윈도우)

```
const char *src_file = "c:/data/test1.txt";
const char *dest_file = "c:/data/mytest1.txt";

/* 해당 파일명의 파일 존재 여부 검사*/
if( _access_s(dest_file, 0) == 0)
{
     if(remove(dest_file) !=0 )
     {
             printf("rename error!\n");
     }
}

if(rename(src_file, dest_file) != 0)
{
     printf("rename error!\n");
}
```

FIO-12. setbuf()함수보다는 setvbuf()함수를 사용하라.

○ setbuf(), setvbuf()함수

파일 스트림이 오픈될 때 버퍼는 자동으로 512바이트의 크기로 할당되며, close시 자동으로 해제된다. 위의 두 함수는 파일 스트립이 사용할 버퍼를 사용자가 지정해 줄 때 사용한다.

```
int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

- stream : 버퍼를 적용할 스트림
- buf : 버퍼로 사용할 메모리 공간을 가리키는 포인터
- type : 버퍼종류
- size : 버퍼의 크기

※ 버퍼로 사용될 공간은 char형 배열일 수 도 있으며 동적할당된 공간일 수 도 있다. 만약 buf의 값이 NULL인경우, size크기로 malloc()함수에 의해 할당되어 사용되어지다 close시 자동으로 해제된다.

만약 버퍼로 사용할 공간을 프로그래머가 직접 동적으로 할당하였다면 close시 직접 해제하여야 한다.

○ 버퍼의 종류

- _IOFBF : full buffer로써 버퍼가 가득 채워져야 버퍼가 비워짐
- _IOLBF : line buffer로써 '\r','\n'의 입력이 있어야 버퍼가 비워짐
- _IONBF : unbuffered로써 버퍼를 사용하지 않는다.

만약 setvbuf()함수의 type의 인자값으로 _IONBF를 지정하는 경우, buf와 size인수는 무시된다.

○ 반환값

성공 : 0
실패 : 0이 아닌 값

○ setbuf()함수는 반환값이 없으므로 버퍼 설정의 성공여부를 확인할 수 없으므로 setbuf()함수보다는 setvbuf()함수를 사용하는것이 좋다.

[예제] setvbuf()함수의 사용예

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

#define BUFSIZE 1024

int main()
{
        const char *file = "data.txt";
        FILE *fp;
        char buf[BUFSIZE];
        char str[128];

        memset(buf, '\0', sizeof(buf));
        fp = fopen(file, "r");
        if (fp == NULL)
        {
                printf("파일오픈실패\n");
                return 1;
        }

        if (setvbuf(fp, buf, buf ? _IOLBF : _IONBF, BUFSIZE) != 0)
        {
                printf("스트림 버퍼 설정 실패\n");
        }
        else
        {
                while (fgets(str, sizeof(str), fp) != NULL)
                {
                        printf("str : %s\n", str);
                        printf("Buf : %s\n", buf);
                }
```

```
        }

        fclose(fp);
        getchar();
        return 0;
}
```

FIO-13. 방금 읽은 한 개의 문자 외의 것을 다시 넣지 마라.

○ ungetc()함수는 해당 스트림에 unsigned int형 문자를 하나 푸쉬백 한다.

○ 여러개를 푸쉬백 할 수는 있으나 파일 위치와 관련된 함수(fseek(), rewind(), fsetpos())를 사용시에는 부쉬백 한 데이터에 대한 보장을 할 수 없다.

```
#include <stdio.h>

int main()
{
        const char *file = "data.txt";
        FILE *fp;
        int len;
        char str[] = "C Programming";
        char ch;


        fp = fopen(file, "w");
        if (fp == NULL)
        {
                printf("파일오픈실패\n");
                return 1;
        }
        len = fwrite(str, 1, sizeof(str)/sizeof(str[0]), fp);
        printf("%d개의 데이터를 출력하였습니다.\n", len);
        fclose(fp);

        fp = fopen(file, "r");
        if (fp == NULL)
        {
                printf("파일오픈실패\n");
                return 1;
        }

        ch = fgetc(fp);
        printf("읽어들인 문자 : %c\n", ch);

        ungetc('#', fp);
        rewind(fp);     // rewind() 호출 이전의 푸쉬백은 보장할 수 없음
        ungetc('?', fp);
        ungetc('$', fp);
```

```
        ch = getc(fp);
        printf("%c\n", ch);
        ch = getc(fp);
        printf("%c\n", ch);
        ch = getc(fp);
        printf("%c\n", ch);

        fclose(fp);
        getchar();
        return 0;
}
```

FIO-14. 파일 스트림에서 텍스트 모드와 바이너리 모드의 차이를 이해하라.

☞ 시스템에 따라 라인넘김(개행)을 나타내는 문자가 틀리다.

Windows	'\r' + '\n'
POSIX	'\n'
Mac	'\r'

☞ 문자에 따른 ASCII코드 값

'\n'	10
'\r'	13
Ctrl + Z	26

☞ 개방모드에 따른 개행문자 기록방식

파일을 오픈할 때 텍스트 모드로 오픈하는 경우, 운영체제에 맞는 형식으로 자동으로 개행정보를 맞추어 준다. 반면 바이너라 모드로 오픈하는 경우에는 개행문자 또한 모두 그대로 읽고 쓰게 된다.

[예제] 바이너리 모드로의 파일 개방

```
#include <stdio.h>

int main()
{
        FILE *fp;
        const char *file = "data.txt";

        char *string[] = {
```

```
                        "Secure",
                        "C",
                        "Programming"
        };
        int i;

        fp = fopen(file, "wb");
        if (fp == NULL)
        {
                printf("파일열기실패!!!\n");
                return 1;
        }

        for (i=0 ; i<sizeof(string)/sizeof(string[0]) ; i++)
        {
                fprintf(fp, "%s\n", *(string+i));
        }

        fclose(fp);
        return 0;
}
```

FIO-31. 동시에 같은 파일을 여러번 열지마라.

하나의 파일을 동시에 여러번 여는것을 금지하는 플랫폼도 있고, 허락하는 플랫폼도 있다.

[문제코드] 아래의 코드는 하나의 파일이 두번 동시에 열리므로 정확한 결과를 기대할 수 없다.

```
#include <stdio.h>

void write_log(const char *);

int main()
{
        FILE *fp;
        const char *file = "log.txt";
        char buf[100];

        char *message = "File Open Test...";

        fp = fopen(file, "r");
        if (fp == NULL)
        {
                printf("파일열기실패!!!\n");
                getchar();
                return 1;
        }

        write_log(message);

        while (fgets(buf, sizeof(buf), fp) != NULL)
        {
                puts(buf);
        }

        fclose(fp);
        getchar();
        return 0;
}

void write_log(const char *message)
{
        FILE *fp;
        char *file = "log.txt";

        fp = fopen(file, "a");
        if (fp == NULL)
        {
                return;
        }

        fputs(message, fp);
        fclose(fp);
}
```

[해결방법] 이미 오픈 된 파일 구조체 포인터를 함수 호출 인자로 전달하여 처리한다.

```
#include <stdio.h>

void write_log(FILE *, const char *);

int main()
{
        FILE *fp;
        const char *file = "log.txt";
        char buf[100];

        char *message = "File Open Test...";

        fp = fopen(file, "r+");
        if (fp == NULL)
        {
                printf("파일열기실패!!!\n");
                getchar();
                return 1;
        }

        write_log(fp, message);

        fseek(fp, 0L, SEEK_SET);
        while (fgets(buf, sizeof(buf), fp) != NULL)
        {
                printf("%s", buf);
        }

        fclose(fp);
        getchar();
        return 0;
}

void write_log(FILE *fp, const char *message)
{
        if (fp == NULL)
        {
                return;
        }

        fseek(fp, 0L, SEEK_END);
        fprintf(fp, "%s\n", message);
}
```

FIO-36. fgets()함수를 사용할 때 개행문자가 읽힌다고 가정하지 마라.
FIO-37. fgets()함수를 사용할 때 문자 데이터를 읽었다고 가정하지 마라.
FIO-40. fgets() 실패 시, 문자열을 초기화 하라.

fgets() 함수는 일반적으로 스트림 입력으로부터 개행문자로 종료된 라인을 읽기 위해 사용된다.

```
char *fgets(char *s, int size, FILE *stream);
```

○ fgets()함수가 스트림으로 부터 입력을 중단하는 3가지 시점

1) 개행문자까지 입력
2) size-1개의 문자까지 입력
3) 파일 끝 표시자 전까지 입력

그러므로 fgets()함수로 입력받은 문자열의 제일 마지막에 개행문자가 있다고 가정할 수 없다.

[문제코드]

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

int main()
{
	FILE *fp;
	const char *file = "log.txt";
	char buf[10];

	fp = fopen(file, "r");
	if (fp == NULL)
	{
		printf("파일열기실패!!!\n");
		return 1;
	}

	while (fgets(buf, sizeof(buf), fp) != NULL)
	{
		/* 읽어들인 문자열의 마지막 문자가 개행문자열이라고 가정 */
		buf[strlen(buf) - 1] = '\0';
		printf("%s\n", buf);

	}

	fclose(fp);
	getchar();
	return 0;
}
```

[해결방법]

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

int main()
{
	FILE *fp;
	const char *file = "log.txt";
	char buf[10];
	char *nr = NULL;

	fp = fopen(file, "r");
	if (fp == NULL)
	{
		printf("파일열기실패!!!\n");
		return 1;
	}

	/* 문자열이 읽혔다고 가정하지 마라 */
	while (fgets(buf, sizeof(buf), fp) != NULL)
	{
		/* 개행문자가 읽힌다고 가정하지 마라 */
		nr = strchr(buf, '\n');
		if (nr != NULL) *nr = '\0';
		printf("%s\n", buf);
	}
	/* 문자열을 초기화 하라 */
	memset(buf, '\0', sizeof(buf));

	fclose(fp);
	getchar();
	return 0;
}
```

FIO-33. 정의되지 않은 동작을 초래하는 입출력 에러를 발견하고 처리하라.

동작이 실패할 때 변수들을 부적절하게 초기화한 상태로 남겨둘 수 있는 입/출력 함수들의 상태를 항상 체크하라.

함수	성공 시 반환 값	실패 시 반환값
fopen()	FILE * 구조체 포인터	NULL 포인터
get()	되도록 사용하지 말것.	
fgets()	문자열의 시작주소	NULL 포인터
sprintf()	되도록 사용하지 말것.	
snprintf()	0 < 반환값 < 버퍼의크기	0 > 반환값 > 버퍼의 크기
vsprintf()	되도록 사용하지 말것.	
vsnprintf()	0 < 반환값 < 버퍼의크기	0 > 반환값 > 버퍼의 크기
strcpy	되도록 사용하지 말것.	
strncpy	복사된 문자열의 시작주소	NULL 포인터

gets()함수는 경계가 불분명한 소스로 부터 고정길이의 배열에 데이터를 복사하는 문제점이 있다. 소스의 길이보다 배열이 짧을 버퍼 오버플로우를 발생시킨다.

[문제코드]

```
#include <stdio.h>
enum { BUFFSIZE=10 };

int main()
{
        char buf[BUFFSIZE];

        gets(buf);
        printf("입력된 문자열 : %s\n", buf);

        getchar();
        return 0;
}
```

[해결방법]

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

enum { BUFFSIZE=10 };

int main()
{
	char *br;
	char buf[BUFFSIZE];
	int ch;

	printf("문자열입력 : ");
	if (fgets(buf, sizeof(buf), stdin))
	{
		br = strchr(buf, '\n');
		if (br)
		{
			*br = '\0';
		}
		else
		{
			while ((ch = getchar()) != '\n' && ch != EOF);
		}

		printf("입력된 문자열 : %s\n", buf);
	}
	else
	{
		printf("데이터 입력오류\n");
		memset(buf, '\0', BUFFSIZE);
	}

	getchar();
	return 0;
}
```

fopen()함수는 실패 시, NULL포인터를 반환하는데 이때 성공여부를 체크하지 않고 파일을 사용하면 프로그램이 다운되거나 의도하지 않은 동작을 일으킨다.

[문제코드]

```c
#include <stdio.h>

int main()
{
	FILE *fp;
	const char *file = "data.txt";
	char *string = "Secure C Programming";

	/* 파일개방이 성공하였는가? */
	fp = fopen(file, "wb");
	fprintf(fp, "%s\n", string);

	fclose(fp);
	return 0;
}
```

[해결방법]

```c
#include <stdio.h>

int main()
{
	FILE *fp;
	const char *file = "data.txt";
	char *string = "Secure C Programming";

	fp = fopen(file, "wb");
	if (fp == NULL)
	{
		/* 에러처리 */
		return 1;
	}

	fprintf(fp, "%s\n", string);
	fclose(fp);

	return 0;
}
```

snprintf()함수는 sprintf()함수에 복사할 문자열의 길이를 지정할 수 있도록 기능이 확장된 함수이다. (지원 안되는 컴파일러도 있음)

인코딩 에러시 -1을 반환하고 결과 값이 버퍼 크기에 맞지 않으면 버퍼 크기 이상의 값을 반환한다.

아래의 코드는 snprintf()함수의 반환값을 고려하지 않아 프로그램의 정상적인 수행을 보장할 수 없다.

[문제코드]

```c
#include <stdio.h>

enum { BUFFSIZE=30 };

int main()
{
        char buffer[BUFFSIZE] = {};
        char s[] = "computer";
        int i = 3000;
        int j = 0;

        /* snprintf()함수의 반환값을 검사하지 않고 사용 */
        j += snprintf(buffer, BUFFSIZE, "String : %s\n", s);
        j += snprintf(buffer+j, BUFFSIZE-j, "String : %d\n", i);

        printf("%s", buffer);

        getchar();
        return 0;
}
```

[해결방법]

```c
#include <stdio.h>

enum { BUFFSIZE=30 };

int main()
{
        char buffer[BUFFSIZE] = {};
        char s[] = "computer";
        int i = 3000;
        int j = 0;
        int rc = 0;

        /* snprintf()함수의 반환값을 검사하지 않고 사용 */
        rc += snprintf(buffer, BUFFSIZE, "String : %s\n", s);
        if (rc == -1){
                printf("문자열 인코딩 실패!!!\n");
                return 1;
```

```
        }
        if (rc >= BUFFSIZE-j)
        {
                printf("버퍼의 공간이 부족합니다.\n");
                return 1;
        }
        else
        {
                j += rc;
        }
        rc += snprintf(buffer+j, BUFFSIZE-j, "String : %d\n", i);
        if (rc == -1){
                printf("문자열 인코딩 실패!!!\n");
                return 1;
        }
        if (rc >= BUFFSIZE-j)
        {
                printf("버퍼의 공간이 부족합니다.\n");
                return 1;
        }

        printf("%s", buffer);

        getchar();
        return 0;
}
```

FIO-34. 문자 I/O 함수의 반환값을 저장할 때에는 int를 사용하라.

fgetc(), getc(), getchar(), fputc(), putc(), putchar(), ungetc() 같은 문자 입출력 함수는 모두 스트림으로부터 문자를 읽어 int로 반환하며, 스트림이 파일 끝에 위치해 있다면 파일 끝 표시자가 설정되어 EOF를 반환한다.

이때 반환값을 저장하는 변수가 char형일 경우, EOF와의 정상적인 비교가 보장되지 않으므로 반환값은 반드시 int형으로 저장하도록 한다.

```
int c;

while ((c = getchar()) != '\n' && c != EOF)
{
        .
        .
        .
}
```

FIO-39. 플러시나 위치조정함수 호출 없이 스트림으로부터 입출력을 교대로 수행하지 마라.

아래의 코드는 파일에 데이터를 붙여쓰기 한 후, 같은 파일로 부터 읽고 있다. 이때 fread() 함수와 fwrite()함수의 호출 사이에서 스트림이 flush되지 않았기 때문에 어떻게 동작할 지 예상할 수 없다.

[문제코드]

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

enum { BUFFSIZE=100 };

int main()
{
        const char *file = "log.txt";
        FILE *fp;
        const char message[BUFFSIZE] = "flush를 해야만이 파일에
기록됩니다.";
        char buf[BUFFSIZE];

        printf("파일을 개방합니다.\n");
        fp = fopen(file, "w+");
        if (fp == NULL)
        {
                printf("파일개방실패!!!\n");
                goto except;
        }

        if (fwrite(message, BUFFSIZE, 1, fp) != 1)
        {
                printf("파일쓰기실패!!!\n");
                goto except;
        }

        memset(buf, '\0', sizeof(buf));
        if (fread(buf, BUFFSIZE, 1, fp) != 1)
        {
                printf("파일읽기실패!!!\n");
                goto except;
        }
        printf("읽어들인 데이터 : %s\n", buf);

except:
        printf("파일을 닫습니다.\n");
        fclose(fp);
        getchar();
        return 0;
}
```

[해결방법]

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

enum { BUFFSIZE=100 };

int main()
{
        const char *file = "log.txt";
        FILE *fp;
        const char message[BUFFSIZE] = "flush를 해야만이 파일에
기록됩니다.";
        char buf[BUFFSIZE];

        printf("파일을 개방합니다.\n");
        fp = fopen(file, "w+");
        if (fp == NULL)
        {
                printf("파일개방실패!!!\n");
                goto except;
        }

        if (fwrite(message, BUFFSIZE, 1, fp) != 1)
        {
                printf("파일쓰기실패!!!\n");
                goto except;
        }

        fseek(fp, 0L, SEEK_SET);

        memset(buf, '\0', sizeof(buf));
        if (fread(buf, BUFFSIZE, 1, fp) != 1)
        {
                printf("파일읽기실패!!!\n");
                goto except;
        }
        printf("읽어들인 데이터 : %s\n", buf);

except:
        printf("파일을 닫습니다.\n");
        fclose(fp);
        getchar();
        return 0;
}
```

제 11장 환경 (ENV)

ENV-00. getenv()함수에서 반환한 문자열을 가리키는 포인터를 저장하지 마라.
ENV-01. 환경변수의 크기를 함부로 가정하지 마라.
ENV-30. getenv()함수가 반환한 문자열을 수정하지 마라.

환경변수의 값이 프로그램 수행 도중에 변경되지 않도록 하여야 하며, 환경변수의 값을 저장할 때에는 저장공간의 길이를 반드시 고려하여야 한다.

getenv()함수가 반환한 포인터 값을 getenv()함수의 호출로 덮어 쓰거나 putenv()함수, setenv()함수 호출 또는 다른 방식을 통해 환경변수의 값에 변경이 발생하면 유효하지 않은 값이 될 수 있으므로 포인터를 저장하는 것은 좋은 방법이 아니다.

[문제코드]

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

enum { BUFFSIZE=100 };

int main()
{
        char *path = getenv("PATH");
        printf("환경변수 PATH의 값\n%s\n", path);
        char *token;
        int i = 0;

        token = strtok(path, ":");
        printf("%2d. %s\n", ++i, token);
        while ((token = strtok(0, ":")) != NULL)
        {
                printf("Dir %2d. %s\n", ++i, token);
        }

        /* token()함수에 의해 환경변수가 변경되었음 */
        printf("token()함수 호출 후 PATH의 값\n%s\n", path);

        getchar();
        return 0;
}
```

[해결방법]

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

enum { BUFFSIZE=100 };

int main()
{
        char *path = getenv("PATH");
        /* 환경변수 값을 복사하여 사용한다. */
        char *tmp = strdup(path);

        if (tmp == NULL)
        {
                printf("환경변수값 복사오류\n");
                return 1;
        }
        printf("환경변수 PATH의 값\n%s\n", path);
        char *token;
        int i = 0;

        token = strtok(tmp, ":");
        printf("%2d. %s\n", ++i, token);
        while ((token = strtok(0, ":")) != NULL)
        {
                printf("Dir %2d. %s\n", ++i, token);
        }

        printf("token()함수 호출 후 PATH의 값\n%s\n", path);

        free(tmp);
        tmp = NULL;
        getchar();
        return 0;
}
```

환경(ENV)

해결방법(윈도우) : _dupenv_s() 함수를 사용하여 환경 변수 문자열의 복사본을 만들어 사용

(윈도우 예제2-1)

```
#include <stdio.h>
#include <malloc.h>
#include <string.h>
#include <Stdlib.h>

int main()
{
    char *tmpvar;
    char *tempvar;
    size_t len;
    /* _dupenv_s() : 환경변수에서 지정된 이름을 탐색하여 버퍼를 할당하고 변수 값을 버퍼에 복사함 - 반환한 버퍼는 꼭 해제해 줄 것*/
    errno_t err = _dupenv_s(&tmpvar, &len, "TMP");
    if(err) return -1;  /* _dupenv_s() 실패 시 */
    err = _dupenv_s(&tempvar, &len, "TEMP");
    if(err)
    {
        free(tmpvar);
        tmpvar = NULL;
        return -1;
    }
```

환경(ENV)

```
    if(strcmp(tmpvar, tempvar) == 0)
    {
        puts("TMP end TEMP are the same.\n");
    }
    else
    {
        puts("TMP end TEMP are NOT the same\n");
    }
    free(tmpvar);
    tmpvar = NULL;
    free(tempvar);
    tempvar = NULL;
```

ENV-02. 이름이 같은 여러개의 환경변수가 존재할 수 있음을 알아두자.

시스템에 따라 여러개의 환경변수가 같은 이름으로 존재할 수 있으므로 프로그램이 일관되게 같은 이름을 선택하도록 하여야 한다.

- 항상 처음 검색된 환경변수의 값을 사용하며 나머지는 무시된다.
- POSIX시스템에서는 환경변수 이름에 대소문자를 구분한다.
- Windows 시스템에서의 환경변수 이름은 대소문자 구분이 없다.

[문제코드] 아래 예제의 결과는 Windows 시스템과 POSIX시스템에서 다르게 나타난다.

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

int main()
{
	char *tmp;

	if (putenv("TEST_ENV=foo") != 0)
	{
		puts("환경변수 TEST_ENV 등록실패\n");
	}
	if (putenv("Test_ENV=bar") != 0)
	{
		puts("환경변수 Test_ENV 등록실패\n");
	}

	tmp = getenv("TEST_ENV");
	if (tmp == NULL)
	{
		puts("환경변수 정보취득 실패\n");
	}
	else
	{
		printf("TEST_ENV : %s\n", tmp);
	}

	getchar();
	return 0;
}
```

[해결방법] 이식성을 고려한다면, 환경변수를 사용할 때, 대소문자의 차이만 둘 것이 아니라 이름 자체가 다른 변수를 사용하도록 한다.

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

int main()
{
        char *tmp;

        if (putenv("TEST_ENV=foo") != 0)
        {
                puts("환경변수 TEST_ENV 등록실패\n");
        }
        if (putenv("OTHER_ENV=bar") != 0)
        {
                puts("환경변수 OTHER_ENV 등록실패\n");
        }

        tmp = getenv("TEST_ENV");
        if (tmp == NULL)
        {
                puts("환경변수 정보취득 실패\n");
        }
        else
        {
                printf("TEST_ENV : %s\n", tmp);
        }

        getchar();
        return 0;
}
```

제12장 에러처리(ERR)

ERR-04. 적절한 종료방법을 선택하라.
ERR-06. assert()함수와 aboart()함수의 종료시 동작을 이해하라.

발견된 오류에 대한 시스템이 대응할 수 있는 두가지 방법

ㅇ fail fast
현재 동작을 바로 실패처리하는것으로 오류에 대한 대응으로 즉시 시스템을 정지 시킨다.

ㅇ fail soft
시스템에서 필수적이지 않은 프로세스들을 종료시켜, 부분적으로라도 서비스나 기능을 제공하는 형태의 시스템으로 시스템의 성능이 떨어진다.

▶ exit() 함수 : 일반적으로 프로그램을 종료할 때 사용

```
void exit(int status);
```

- 정상종료 : EXIT_SUCCESS(0)을 인자로 사용
- 비정상종료 : EXIT_FAILURE(1)을 인자로 사용

- 아직 쓰여지지 않은 버퍼에 남아 있는 데이터를 flush한다.
- 열려 있는 모든 파일을 닫는다.
- 임시 파일을 지운다.
- 운영체제에게 정수로 된 종료 상태값을 반환한다.

▶ atexit() 함수 : 프로그램 종료 시, 부가적인 동작을 수행하기 위해 사용하는 함수로써 atexit() 함수를 이용하여 함수를 등록하면 프로그램 종료 시, 등록해놓은 함수가 수행된다.

```
void atexit(void (*)(void));
```

[예제] exit()함수와 atexit()함수

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

void my_exit(void)
{
        printf("my_exit() 호출\n");
}

int main()
{
        atexit(my_exit);
        exit(EXIT_SUCCESS);

        printf("main() 함수 실행\n");
        return 0;
}
```

▶ abort() 함수 : 가장 빠르고 즉각적으로 운영체제에게 프로그램의 비정상적인 종료를 요청한다.

※ assert() 함수는 내부적으로 abort() 함수를 호출하여 프로그램을 종료시킨다.

▶ 프로그램 종료시의 동작

함수	파일 디스크립터 닫기	버퍼 플러시	임시파일 제거	atexit() 함수호출
abort()	정의되지 않음	정의되지 않음	정의되지 않음	호출안함
_Exit(status)	닫음	정의되지 않음	정의되지 않음	호출안함
exit(status)	닫음	플러시함	제거함	호출함
main()에서 retur 문에 의해 종료	닫음	플러시함	제거함	호출함

종료하기 전에 지정한 중요한 동작을 수행해야 하는 경우에는 abort() 함수로 종료하는것은 적절하지 않다.

[문제코드] 다음 프로그램은 데이터가 파일에 저장되었는지의 여부가 불확실하다.

```c
#include <stdio.h>
#include <stdlib.h>
#define FILENAME "log.txt"

int write_data();

int main()
{
	if (write_data())
	{
		printf("파일에 데이터를 기록하였습니다.\n");
	}

	return 0;
}

int write_data()
{
	const char *file = FILENAME;
	FILE *fp = fopen(file, "w");
	if (fp == NULL)
	{
		printf("파일오픈에러\n");
		return 0;
	}

	fprintf(fp, "Hello, World\n");
	abort();

	fclose(fp);
	return 1;
}
```

[해결방법] abort()가 아닌 exit()함수를 호출하여 종료한다.

```c
#include <stdio.h>
#include <stdlib.h>
#define FILENAME "log.txt"

int write_data();

int main()
{
	if (write_data())
	{
		printf("파일에 데이터를 기록하였습니다.\n");
	}

	return 0;
}
```

```
int write_data()
{
        const char *file = FILENAME;
        FILE *fp = fopen(file, "w");
        if (fp == NULL)
        {
                printf("파일오픈에러\n");
                return 0;
        }

        fprintf(fp, "Hello, World\n");
        exit(EXIT_FAILURE);

        fclose(fp);
        return 1;
}
```

ERR-30. errno를 사용하는 라이브러리 함수를 호출하기 전에는 errno의 값을 0으로 설정하고, 함수가 에러를 의미하는 값을 반환 했을 때, errno의 값을 확인하라.

ㅇ 일반적으로 함수 수행에 대한 에러를 감지하는 것은 반환 값을 확인하며, errno는 단지 해당 에러의 원인이 무엇인지를 파악할 때 만 사용한다.

▶ errno 사용 시 주의할 점

ㅇ 프로그램 시작 시 errno의 값은 0으로 시작된다.

ㅇ 특정 라이브러리 함수(errno 를 설정하는 함수) 수행 시, 에러가 발생되면 errno의 값이 0이 아닌 값으로 설정 됨

ㅇ errno를 다시 사용하고자 할 때에는 반드시 errno의 값을 0으로 설정해 주어야 한다. (어떤 라이브러리 함수도 이것을 대신 수행하지 않는다)

[문제코드] strtoul() 함수를 호출하기 전에 errno 를 0으로 초기화 하지 않았다.

```c
#include <stdio.h>
#include <stdio_ext.h>
#include <stdlib.h>
#include <errno.h>
#include <limits.h>

int main()
{
        char buf[100];
        unsigned long number;
        char *endptr;

        printf("정수입력 (%lu ~ %lu) : ", 0L, ULONG_MAX);
        fgets(buf, sizeof(buf), stdin);

        number = strtoul(buf, &endptr, 10);

        if (buf == endptr && (*endptr == '\n' || *endptr == '\0'))
        {
                printf("입력된 문자가 없습니다.\n");
        }
        else if (*endptr != '\n' && *endptr != '\0')
        {
                printf("변환할 수 없는 문자가 포함되어 있습니다.\n");
        }
        else if (errno == ERANGE)
        {
                printf("long형 범위 밖의 숫자가 입력되었습니다.\n");
        }
        else
        {
```

```
                printf("입력된 값 : %lu\n", number);
        }

        __fpurge(stdin);
        getchar();
        return 0;
}
```

[해결방법] errno를 사용하기 전에 반드시 0으로 초기화 하라.

```
#include <stdio.h>
#include <stdio_ext.h>
#include <stdlib.h>
#include <errno.h>
#include <limits.h>

int main()
{
        char buf[100];
        unsigned long number;
        char *endptr;

        printf("정수입력 (%lu ~ %lu) : ", 0L, ULONG_MAX);
        fgets(buf, sizeof(buf), stdin);

        errno = 0;
        number = strtoul(buf, &endptr, 10);

        if (buf == endptr && (*endptr == '\n' || *endptr == '\0'))
        {
                printf("입력된 문자가 없습니다.\n");
        }
        else if (*endptr != '\n' && *endptr != '\0')
        {
                printf("변환할 수 없는 문자가 포함되어 있습니다.\n");
        }
        else if (errno == ERANGE)
        {
                printf("long형 범위 밖의 숫자가 입력되었습니다.\n");
        }
        else
        {
                printf("입력된 값 : %lu\n", number);
        }

        __fpurge(stdin);
        getchar();
        return 0;
}
```

▶ 라이브러리 함수와 errno

○ 라이브러리 함수가 errno를 설정하지만 모호한 에러 식별자를 반환하는 경우, 즉 에러 상태의 반환값과 성공 시 반환값이 구분되지 않는 경우에 errno를 사용한다.

○ 라이브러리 함수가 errno를 설정하지만 함수의 반환값이 명확할 때
함수의 반환값을 이용해 함수 실행의 성공여부를 먼저 확인한 후, 실패 일 경우에 errno의 값을 체크한다.

○ errno를 설정한다고 보장할 수 없는 라이브러리 함수인 경우와 라이브러리 함수가 표준과 다르게 문서화 되어있는 경우에는 errno에 대한 확실한 보장이 없으므로 errno를 사용하지 않도록 한다.

[문제코드] calloc() 함수의 반환값을 확인하여 함수의 성공/실패여부를 확인하지 않은 상태에서 errno를 체크하고 있다.

```c
#include <stdio.h>
#include <stdio_ext.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

enum { STRING_SIZE = 100 };

int main()
{
        char *name = "Secure C Programming";
        char *str;

        if (strlen(name) >= STRING_SIZE)
        {
                printf("문자열이 너무 길어서 복사가 불가능 합니다.\n");
                return 1;
        }

        errno = 0;
        str = (char *)calloc(STRING_SIZE, sizeof(char));
        /* calloc() 함수의 반환값을 확인하지 않고 errno를 체크하고 있다. */
        if (errno == ENOMEM)
        {
                printf("메모리 부족으로 인해 동적메모리 할당에
실패하였습니다.\n");
                return 1;
        }

        strncpy(str, name, strlen(name));
        printf("복사된 문자열 : %s\n", str);
        free(str);
```

```
        str = NULL;

        getchar();
        return 0;
}
```

[해결방법] 함수의 반환값을 먼저 확인하도록 한다.

```
#include <stdio.h>
#include <stdio_ext.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

enum { STRING_SIZE = 100 };

int main()
{
        char *name = "Secure C Programming";
        char *str;

        if (strlen(name) >= STRING_SIZE)
        {
                printf("문자열이 너무 길어서 복사가 불가능 합니다.\n");
                return 1;
        }

        errno = 0;
        str = (char *)calloc(STRING_SIZE, sizeof(char));
        /* calloc() 함수의 성공여부를 우선 확인한다. */
        if (str == NULL)
        {
                printf("동적메모리할당실패!!!\n");
                /* 실패하였을 경우, errno를 체크한다. */
                if (errno == ENOMEM)
                {
                        printf("ERRNO : %d\n", errno);
                        return 1;
                }
        }

        strncpy(str, name, strlen(name));
        printf("복사된 문자열 : %s\n", str);
        free(str);
        str = NULL;

        getchar();
        return 0;
}
```

부록 #1. 함수

1. 입력함수

▶ scanf() 계열의 함수

헤더	함수원형
stdio.h	int scanf(const char *format, ...);
stdio.h	int fscanf(FILE *fp, const char *format, ...);
stdio.h	int sscanf(const char *buffer, const char format, ...);
stdio.h	int vscanf(const char *format, va_list arglist);
stdio.h	int vfscanf(FILE *fp, const char *format, va_list arglist);
stdio.h	int vsscanf(const char *buffer, const char *format, va_list arglist);

◑ scanf() 함수

<table>
<tr><td>기 능</td><td colspan="2">format 형태대로 stdin(표준입력)으로부터 입력받는다.</td></tr>
<tr><td>헤 더</td><td colspan="2">#include <stdio.h></td></tr>
<tr><td>선 언</td><td colspan="2">int scanf(const char *format, ...);</td></tr>
<tr><td rowspan="2">형식인자</td><td>format</td><td>입력받을 서식</td></tr>
<tr><td>...</td><td>입력받을 인수들로써 주소값을 전달하여야 한다.</td></tr>
<tr><td rowspan="2">리 턴</td><td>성공시</td><td>입력받은 형식 갯수</td></tr>
<tr><td colspan="2">입력받은것이 없을 경우에는 0을 반환함</td></tr>
<tr><td>설 명</td><td colspan="2">scanf()계열 함수들을 사용할 때 주의할 것은, 인수를 사용할 때 인수의 주소값을 전달하여야 한다는 것이다. 즉「int a;」라고 선언된 변수에 값을 입력받기 위해 scanf()함수를 사용한다면 아래와 같은 방법으로 변수 a의 주소를 전달하여야 한다.

scanf("%d", &a);

scanf()함수를 실행하여 키보드로부터 입력받을 때 입력한 내용은 화면으로 출력되며 (이를 echo된다라고 함) 다중 입력의 경우 각각의 입력항목을「Space Bar」나「Enter」키로 구분하고, 마지막 항목을 입력하고 난 뒤「Enter」키를 입력치면 입력을 종료한다.</td></tr>
</table>

◑ **fscanf() 함수**

기 능	format 형태대로 파일로 부터 입력받는다.	
헤 더	#include <stdio.h>	
선 언	int fscanf(FILE *fp, const char * format, ...);	
형식인자	fp	데이터를 입력받을 파일을 가리키는 FILE 구조체 포인터
	format	입력받을 서식
	...	입력된 값을 저장할 인수들
리 턴	성공시	입력된 서식의 수
	문자열의 끝을 입력받은 경우 매크로 EOF를 반환함	
설명		

◑ **sscanf() 함수**

기 능	format 형태대로 문자열로부터 입력받는다.	
헤 더	#include <stdio.h>	
선 언	int sscanf(const char *buffer, const char *format, ...);	
형식인자	buffer	읽어들일 문자열을 가리키는 포인터
	format	입력받을 서식
	arglist	가변개수 인수들을 가리키는 포인터
리 턴	성공시	출력한 문자들의 바이트 수
	입력받은것이 없을 경우에는 0을 반환함	
설 명		

◑ **vscanf() 함수**

기 능	format 형태대로 stdin(표준입력)으로 부터 입력받는다.	
헤 더	#include <stdio.h>	
선 언	int vscanf(const char *format, valist_ args);	
형식인자	format	입력받을 서식
	arglist	입력된 데이터를 저장할 가변인수를 가리키는 포인터
리 턴	성공시	출력한 문자들의 바이트 수

	입력받은것이 없을 경우에는 0을 반환함	
설 명		

◑ **vfscanf() 함수**

기 능	format 형태대로 파일로부터 입력받는다.	
헤 더	#include <stdio.h>	
선 언	int vfscanf(FILE *fp, const char *format, va_list arglist);	
형식인자	fp	데이터를 입력받을 파일을 가리키는 FILE구조체 포인터
	format	입력받을 서식
	arglist	입력된 데이터를 저장할 가변개수 인수를 가리키는 포인터
리 턴	성공시	입력된 서식의 갯수
	문자열의 끝을 입력받은 경우 EOF를 반환 함	
설 명		

◑ **vsscanf() 함수**

기 능	format 형태대로 문자열로부터 입력받는다.	
헤 더	#include <stdio.h>	
선 언	int vsscanf(const char *buffer, const char *format, va_list arglist);	
형식인자	buffer	읽어들일 문자열을 가리키는 포인터
	format	입력받을 서식
	arglist	입력된 데이터를 저장할 가변개수 인수를 가리키는 포인터
리 턴	성공시	입력된 서식의 갯수
	입력받은 값이 없을 경우 0을 반환 함	
설 명		

※ scanf()계열함수들의 문제점

다양한 용도에서 사용할 수 있는 scanf()계열 함수를 사용하였을 때의 가장 큰 문제점은 방향키와 같은 특수한 용도로 사용되는 키의 입력은 받을 수 없다는 것이다.

▶ **문자입력함수**

헤더	함수원형
stdio.h	int fgetc(FILE *fp);
stdio.h	int getc(FILE *fp);
stdio.h	int getchar(void);

◑ **fgetc() 함수**

기 능	문자 하나를 파일로부터 입력받는다.	
헤 더	#include <stdio.h>	
선 언	int fgetc(FILE *fp);	
형식인자	fp	문자를 읽어들일 파일을 가리키는 포인터
리 턴	성공시	입력된 문자의 ASCII 코드값
	실패시	매크로 EOF를 반환
설 명	이 함수는 지정한 파일로부터 한 문자를 입력받고, 그 문자의 아스키 코드값을 반환한다.	

◑ **getc() 함수**

기 능	문자 하나를 파일로부터 입력받는다.	
헤 더	#include <stdio.h>	
선 언	int getc(FILE *fp);	
형식인자	fp	문자를 읽어들일 파일을 가리키는 포인터
리 턴	성공시	입력된 문자의 ASCII 코드값
	실패시	매크로 EOF를 반환
설 명	fgetc()함수와 같은 기능을 수행하며 사용법도 동일하다.	

◑ **getchar() 함수**

기 능	문자 하나를 stdin(표준입력)으로부터 입력받는다.	
헤 더	#include <stdio.h>	
선 언	int getchar();	
리 턴	성공시	입력된 문자의 ASCII 코드값

	실패시	매크로 EOF를 반환
설 명	표준입력이란 키보드를 말한다. 즉 이 함수는 키보드로부터 하나의 문자를 입력받고 입력받은 문자의 아스키 코드값을 반환하는 함수이다. 키보드로 입력된 문자는 다시 화면에 echo되며 입력의 종룔르 나타내키 위해서 「Enter」키를 사용한다.	

▶ 기능키의 입력

컴퓨터의 키보드는 단순히 문자만을 입력하는 장치가 아니다. 각 키에 따라서 문자가 입력될 수 도 있으며 또는 여러가지 기능을 수행하게 되는데 이 기능의 대표적인 예가 방향키이다. 방향키는 커서를 이동시키고 「Enter」키는 입력된 명령을 실행시킨다. 이러한 기능은 키보드 자체에서나 히드웨어에서 제공하는 것이 아니며 프로그램 내에서 키보드로 입력한 값을 판단한 후, 그 값이 방향키이면 커서의 이동을 「Enter」이면 명령을 실행하게끔 하는 것이다.

이는 프로그래머가 입력키에 대한 프로그램의 동적을 어떻게 설계하는가에 따라 달라진다. 예를들어 화살표 키가 아닌 「i, j, k, m」등의 키를 눌렀을 때 커서를 이동하도록 프로그래밍을 하였다면 프로그램은 「i, j, k, m」키를 눌렀을 때 문자의 입력이 아닌 커서를 이동시키게 되는 것이다.

그렇다면 이러한 기능을 가지는 키를 입력받기 위해서는 어떻게 해야 하는가?

▶ 입력 함수의 선택

기능키를 입력받을 때 고려할 사항은 다음과 같은 두가지가 있다.

첫째. 「Enter」키를 눌러야 입력이 이뤄져서는 안된다.
둘째. 기능키를 입력받았을 때 화면에 echo되지 말아야 한다.

scanf()계열의 함수는 원하는 데이터를 입력한 후, 「Enter」키를 눌러야만 데이터 입력이 이뤄진다. 그러므로 기능키를 입력받는 함수로써는 부적절하다. 예를들어 커서를 이동시키기 위해 화살표키를 입력받는다고 가정할 때 화살표키를 누른 후, 입력의 종료를 나타내기 위해 「Enter」키를 매번 눌러야만 한다면 어떨까? 그러므로 이러한 기능키를 입력받기 위해서는 입력의 종료를 나타내기 위해 「Enter」키를 눌러야만 하는 함수들은 부적절 하다. 뿐만아니라 기능키를 입력받았을 때에는 입력된 값이 화면에 echo 되지 말아야 한다.

▶ 입력버퍼

컴퓨터에는 입력버퍼라는것이 있다. 키보드로부터 또는 파일로부터 데이터를 읽어들임에 있어서 그 값이 바로 변수에 전달되는 것이 아니라 이 버퍼라는곳으로 입력된 후, 버퍼로부터 값을 읽어들인다. 즉 입력함수들은 키보드로부터 또는 파일로부터 데이터를 직접 입력받는것이

아니라 키보드로부터 입력된 데이터 또는 파일로부터 읽어들인 데이터가 우선 버퍼로 저장되고 이 버퍼로부터 데이터를 읽어들이게 되는 것이다.

예를들어 키보드를 누르면 그 키에 해당하는 아스키 코드값이 입력버퍼에 저장되고, scanf()나 fgetc()함수들을 사용하면 이 버퍼에 저장되어 있는 값들을 읽어들여 변수에 저장하게 된다.

```
char value = fgetchar();
```

만약 위와 같은 명령문을 사용하였을 때 fgetchar()함수는 입력버퍼를 검사하여 저장된 값이 없으면 입력을 기다리게 되고 저장된 값이 있다면 입력버퍼로부터 1바이트 크기를 읽어들여 value에 입력한다. 여기서 주의할 것은 읽어들이는 데이터의 크기는 1바이트라는 것이다. 즉 입력버퍼에 1바이트 이상의 값들이 저장되어 있다면 읽어들이고 난 후, 입력버퍼상에 남아있는 나머지 데이터를 모두 읽어들여야만 이후 새로운 데이터를 입력받을 수 있게된다.

▶ 문자열 입력함수

헤더	함수원형
stdio.h	char *gets(char *buffer);
stdio.h	char *fgets(char *buffer, int n, FILE *fp);

◑ gets() 함수

기 능	문자열을 stdin(표준입력)으로부터 입력받는다.	
헤 더	#include <stdio.h>	
선 언	char *gets(char *buffer);	
형식인자	buffer	입력받은 문자열을 저장할 포인터
리 턴	성공시	문자열을 가리키는 포인터
	실패시	매크로 NULL을 반환
설 명	키보드로부터 문자열을 입력받는 함수이다. 키보드로 문자열을 입력한 후, 「Enter」키를 누르면 입력이 종료되며 「Enter」키의 값은 「'\0'」(널문자)로 변환되어 저장된다. 따라서 buffer는 적어도 입력받고자 하는 문자의 갯수 + 1개 만큼의 크기이어야만 한다. 만약 입력받은 문자열을 저장할 buffer의 길이가 입력된 문자열의 갯수 + 1개 보다 작은 경우 다른 변수의 영역을 침범할 우려가 있다. 또한 scanf()함수의 경우 문자열을 입력받을 때 공백문자는 입력받을 수 없지만 gets()함수는 공백문자도 입력받을 수 있다. 또한 gets()함수는 「Ctrl + z」가 입력되면 NULL을 리턴한다.	

◑ **fgets() 함수**

기 능	문자열을 file로부터 입력받는다.	
헤 더	#include <stdio.h>	
선 언	char *fgets(char *buffer, int n, FILE *fp);	
형식인자	buffer	입력받은 문자열을 저장할 포인터
	n	읽어들일 문자열의 길이
	fp	읽어들일 파일을 가리키는 포인터
리 턴	성공시	buffer를 가리키는 포인터
	실패시	매크로 NULL을 반환
설 명	지정한 파일로부터 n-1개 만큼 문자들을 읽어들여 buffer에 저장한다. buffer의 끝에는「'\0'」(널문자)를 추가한다. 이때 읽어들인 문자가 만약 파일의 끝인 EOF라면 입력을 중단하며 EOF는 문자열 s에「'\0'」(널문자)로 변환되어 저장된다. fgets()함수는 입력받을 문자열의 길이를 지정할 수 있으나 지정한 문자열의 갯수에 도달하지 않더라도「'\n'」(개행문자)를 만나면 입력을 중단한다.「'\n'」(개행문자)는「'\0'」(널문자)로 변환되지 않고 그대로 저장되며, 그 대신 s의 끝에「'\0'」(널문자)를 추가한다.	

2. 출력함수

▶ printf() 계열의 함수

헤더	함수원형
stdio.h	int printf(const char *format, ...);
stdio.h	int fprint(FILE *fp, const *format, ...);
stdio.h	int sprintf(char *buffer, const char *format, ...);
stdio.h	int vprintf(const char *format, va_list arglist);
stdio.h	int vfprintf(FILE *fp, const char *format, va_list arglist);
stdio.h	int vsprintf(char *buffer, const char *format, va_list arglist);
stdio.h	int snprintf(char *buffer, size_t size, const char *format, ...);
stdio.h	int vsnprintf(char *buffer, size_t size, const char *format, va_list arglist);

◑ printf() 함수

기 능	format 형태대로 stdout(표준출력)에 출력한다.	
헤 더	#include <stdio.h>	
선 언	int printf(const char *format, ...);	
형식인자	format	출력할 때 의 서식
	...	인수들
리 턴	성공시	출력한 문자들의 바이트 수
	실패시	EOF
설 명	printf() 함수에서의 stdout(표준출력)이란 화면으로의 출력을 말한다. 인수 format은 출력할 때의 서식을 지정하는데 다음의 세가지를 혼용하여 사용한다. 1. 일반문자열 : 문자열을 출력 예) printf("I am a boy"); 2. 출력 변환 기호 : 지정한 인수의 값을 출력 예) printf("%d %d", value1, value2); 3. 확장열 : 특수한 기능을 가지는 Escape 문자 예) printf("\n");	

◑ **fprintf() 함수**

기 능	format 형태대로 파일에 출력한다.	
헤 더	#include <stdio.h>	
선 언	int fprintf(const char *format, ...);	
형식인자	fp	출력할 파일을 가리키는 FILE 구조체 포인터
	format	출력할 때 의 서식
	...	인수들
리 턴	성공시	출력한 문자들의 바이트 수
	실패시	EOF
설 명	파일에 내용을 출력하기 위해선 단지 파일 출력 함수인 fprintf() 함수만을 사용하는 것이 아니라, fopen()이나 fclose() 함수와 같이 파일을 열거나 닫는 함수를 같이 사용하여야 한다.	

◑ **sprintf() 함수**

기 능	format 형태대로 문자열을 생성한다.	
헤 더	#include <stdio.h>	
선 언	int sprintf(char *buffer, const char *format, ...);	
형식인자	buffer	생성된 문자열을 저장할 공간을 가리키는 포인터
	format	출력할 때 의 서식
	...	인수들
리 턴	성공시	출력한 문자들의 바이트 수
	실패시	EOF
설 명	sprintf() 함수의 사용법은 printf() 함수와 비슷하다. 차이점이라면 출력의 결과가 stdout이 아닌 첫번째 파라미터로 주어진 포인터가 가리키는 위치에 저장된다는 것이다. 이때 새로 생성되는 문자열의 길이가 생성된 문자열을 저장할 공간의 크기보다 큰 경우 버퍼오버런(Buffer over run) 현상이 발생될 수 있다.	

◑ **vprintf() 함수**

기 능	format 형태대로 stdout(표준출력)에 출력한다.
헤 더	#include <stdio.h>

<table>
<tr><td>선 언</td><td colspan="2">int vprintf(const char *format, va_list arglist);</td></tr>
<tr><td rowspan="2">형식인자</td><td>format</td><td>출력할 때 의 서식</td></tr>
<tr><td>arglist</td><td>가변인수를 가리키는 포인터</td></tr>
<tr><td rowspan="2">리 턴</td><td>성공시</td><td>출력한 문자들의 바이트 수</td></tr>
<tr><td>실패시</td><td>EOF</td></tr>
<tr><td>설 명</td><td colspan="2">vprintf()함수의 사용법과 기능은 printf()함수와 똑같지만 가변인수를 사용하는 방법이 다르다. printf()함수는「...」을 사용하여 가변 개수 인수들을 하나 하나 전달받는 방식이지만 vprintf()함수는 가변인수들을 가리키는 포인터인 arglist를 사용한다.

[예제] 가변인수의 사용예

<pre>
#include <stdio.h>
#include <stdarg.h>

int print(char *format, ...)
{
        va_list ap;
        int cnt;

        va_start(ap, format);
        cnt = vprintf(format, ap);
        va_end(ap);

        return(cnt);
 }

int main(void)
{
        int i = 30;
        double d = 90.0;
        char *str = "Hello!!!";
        int cnt = 0;

        cnt = print("%d %.2lf %s\n", i, d, str);
        printf("%d개의 문자가 출력되었습니다.\n", cnt);

        getchar();
        return 0;
}
</pre>
</td></tr>
</table>

◑ **vfprintf() 함수**

기 능	format 형태대로 문자열을 파일에 저장한다.	
헤 더	#include <stdio.h>	
선 언	int vfprintf(FILE *fp, const char *format, va_list arglist);	
형식인자	fp	출력할 파일을 가리키는 FILE구조체 포인터
	format	출력할 때 의 서식
	arglist	가변개수 인수를 가리키는 포인터
리 턴	성공시	출력한 문자들의 바이트 수
	실패시	EOF
설 명	vfprintf() 함수의 사용법은 fprintf() 함수와 비슷하다. 차이점이라면 출력의 결과가 stdout이 아닌 첫번째 파라미터로 주어진 포인터가 가리키는 파일에 저장된다는 것이다. 또한 fprintf()함수는 「...」을 사용하여 가변 개수 인수들을 하나 하나 전달받는 방식이지만 vfprintf()함수는 가변인수들을 가리키는 포인터인 arglist를 사용한다.	

◑ **vsprintf() 함수**

기 능	format 형태대로 문자열을 파일에 저장한다.	
헤 더	#include <stdio.h>	
선 언	int vsprintf(char *buffer, const char *format, va_list arglist);	
형식인자	buffer	생성된 문자열을 저장할 공간을 가리키는 포인터
	format	출력할 때 의 서식
	arglist	가변개수 인수를 가리키는 포인터
리 턴	성공시	출력한 문자들의 바이트 수
	실패시	EOF
설 명	vsprintf()함수는 sprintf()함수와 같은 기능을 수행한다. 차이점은 sprintf() 함수는 「...」을 사용하여 가변 개수 인수들을 하나 하나 전달받는 방식이지만 vsprintf() 함수는 가변인수들을 가리키는 포인터인 arglist를 사용한다.	

◑ **snprintf() 함수**

기 능	format 형태대로 문자열을 파일에 저장한다.	
헤 더	#include <stdio.h>	
선 언	int snprintf(char *buffer, size_t size, const char *format, ...);	
형식인자	buffer	생성된 문자열을 저장할 공간을 가리키는 포인터
	size	buffer에 저장할 문자의 수
	format	출력할 때 의 서식
	...	인수목록
리 턴	성공시	출력한 문자들의 바이트 수
	실패시	EOF
설 명	snprintf()함수는 sprintf()함수처럼 새로운 문자열을 생성하여 첫번째 파라미터로 주어진 포인터가 가리키는 위치에 문자열을 저장한다. 차이점은 sprintf()함수는 저장할 문자의 갯수를 지정할 수 없는 반면 snprintf()함수는 저장할 문자의 수를 지정할 수 가 있다. 그러므로 snprintf()함수를 이용하는 경우 생성된 문자의 수가 저장할 공간의 크기보다 커서 발생할 수 있는 Buffer over run 현상을 방지할 수 있다는 장점이 있다.	

◑ **vsnprintf() 함수**

기 능	format 형태대로 문자열을 파일에 저장한다.	
헤 더	#include <stdio.h>	
선 언	int vsnprintf(char *buffer, size_t size, const char *format, va_list arglist);	
형식인자	buffer	생성된 문자열을 저장할 공간을 가리키는 포인터
	size	buffer에 저장할 문자의 수
	format	출력할 때 의 서식
	arglist	가변개수 인수를 가리키는 포인터
리 턴	성공시	출력한 문자들의 바이트 수
	실패시	EOF
설 명	vsnprintf()함수는 snprintf()함수와 비슷한 기능을 수행한다. 차이점은 차이점은 snprintf()함수는 「...」을 사용하여 가변 개수	

	인수들을 하나 하나 전달받는 방식이지만 vsnprintf()함수는 가변인수들을 가리키는 포인터인 arglist를 사용한다.

▶ 가변인수

가변인수란 개수가 일정치 않은 인수들을 말한다. 예들들어 printf()함수나 scanf()함수들이 가변인수를 사용하는 함수들이다.

예) printf("%d %d %s \n", i, j, str);

위의 예에서 「"%d %d %s", i, j, str」가 인수이다. 하지만 printf()함수는 첫번째 파라미터로 주어진 출력문자열 내에 있는 출력형식의 갯수 만큼 출력 인수를 지정하여야 하므로 인수가 1개일 수 도 있고 1개 이상 여러개 일 수 도 있다. 이처럼 개수가 일정치 않은 인수를 가리켜 가변인수라고 한다.

◑ 가변인수의 사용

가변인수를 사용하기 위해서는「stdarg.h」헤더를 include 하여야 한다.「stdarg.h」헤더에는 다음과 같은 매크로 함수가 정의되어 있다.

```
typedef void *va_list;
va_start(ap, pramN)
va_arg(ap, type)
va_end(ap)
```

① va_list ap

가변인수를 가리키는 포인터이다. 즉 va_list variable; 이라고 변수를 선언하면 variable은 가변인수들을 가리킬 수 있는 포인터형 변수가 된다.

함수로 전달되는 인수들은 stack이라는 기억장소에 저장되며 함수는 이 stack에 저장된 인수를 읽어들여서 사용한다. 스택에 있는 인수를 읽을 때 포인터 연산을 하여야 하는데 현재 읽고 있는 번지를 기억하기 위한 va_list형의 포인터 변수가 필요하다.

※ va_list 형은 char * 형으로 정의되어 있다.

② va_start(ap, 마지막고정인수)

가변인수의 값을 읽어들이기 위한 준비를 하는 함수로써 ap가 첫번째 가변인수를 가리키도록 초기화 한다. 첫번째 가변인수의 번지를 조사하기 위해서 마지막 고정인수를 전달해 주어야 한다. va_start()함수는 마지막 고정 인수 다음 번지로 ap의 주소를 맞추어 준다.

```
int print(char *format, ...)
{
        va_list ap;
        int cnt;

        va_start(ap, format);
        cnt = vprintf(format, ap);
        va_end(ap);
```

```
        return(cnt);
    }
```

위의 예제에서 va_start(ap, format); 함수의 호출로 인해 ap는 print()함수 호출시 주어진 인수중 마지막 고정인수인 format 다음의 인수(가변인수)의 시작주소를 가리키게 된다.

③ va_arg(ap, 인수타입)

가변인수를 실제로 읽어들이는 함수이다. 이때 가변인수의 시작위치는 va_start()함수에 의해 지정되므로 값일 읽어들이기만 하면 되지만 이때 읽어들일 값의 자료형을 알아야 한다. 그러므로 두번째 인수로써 가변인수로 부터 읽어들일 값의 자료형을 지정한다.

va_arg()함수에서 두번째 인자로 자료형을 지정할 수 있는 이유는 va_arg()함수는 매크로 함수이기 때문이다. 즉 va_arg()함수는 첫번째 인자로 주어진 ap로부터 두번째 인자로 주어진 데이터형으로 데이터를 읽어들이는 역할 뿐만이 아니라 읽어들인 데이터 이후의 데이터를 계속 읽어들이기 위해 두번째 인자로 주어진 자료영의 크기만큼 포인터연산을 하여 주소를 이동시킨다. 그러므로써 다음 인수를 읽어들일 수 있도록 한다.

④ va_end(ap)

va_end()함수는 가변인수를 모두 읽어들인 후, 뒷정리를 하는 함수이지만 별다른 동작은 하지 않으며 실제로 없어도 전혀 지장은 없는 함수이다. 그럼에도 이 명령이 필요한 이유는 호환성때문인데 플랫폼에 따라서는 가변인수를 읽은 후에 뒷처리를 해주어야할 필요가 있기 때문이다.

적어도 인텔계열의 CPU에서는 va_end()함수는 아무런 동작도 하지 않는다. 하지만 다른 플랫폼이나 다른 환경에서 va_end()함수가 중요한 역할을 할 수 도 있으므로 가변인수를 사용하고난 후에는 va_end()함수를 호출하는것이 좋다.

◑ 가변함수의 사용조건

가. 가변인수를 사용하는 함수는 반드시 하나 이상의 고정인수를 가져야만 한다.

나. 가변인수를 사용하기 위해서는 인수의 끝을 알수 있도록 하여야 한다.

다. 가변인수를 사용하기 위해서는 가변인수의 자료형을 알아야만 한다.

▶ **문자출력함수**

헤더	함수원형
stdio.h	int fputc(int ch, FILE *fp);
stdio.h	int fputchar(int ch);
stdio.h	int putc(int ch, FILE *fp);
stdio.h	int putchar(int ch);

◑ **fputc() 함수**

기 능	문자 하나를 file에 출력한다.	
헤 더	#include <stdio.h>	
선 언	int fputc(int ch, FILE *fp);	
형식인자	ch	출력할 문자의 ASCII코드 값
	file	문자를 출력할 파일을 가리키는 FILE구조체 포인터
리 턴	성공시	출력한 문자의 ASCII코드값
	실패시	EOF
설 명	출력할 문자를 나타내는 인수「ch」에 아스키 코드값이나 문자상수를 지정하면 아스키 코드값에 해당하는 문자나 해당 문자가「fp」가 가리키는 파일에 출력된다.	

◑ **fputchar() 함수**

기 능	문자 하나를 stdout(표준출력)에 출력한다.	
헤 더	#include <stdio.h>	
선 언	int fputchar(int ch);	
형식인자	ch	출력할 문자의 ASCII코드 값
리 턴	성공시	출력한 문자의 ASCII코드값
	실패시	EOF
설 명	표준출력이란 화면을 말한다. 즉 이 함수는 화면에 인수로 주어진 아스키 코드값에 해당하는 문자나 인수로 주어진 문자를 출력한다.	

◑ **putc() 함수**

기 능	문자 하나를 file에 출력한다.	
헤 더	#include <stdio.h>	
선 언	int putc(int ch, FILE *fp);	
형식인자	int ch	출력할 문자의 ASCII코드 값
	file	문자를 출력할 파일을 가리키는 FILE구조체 포인터
리 턴	성공시	출력한 문자의 ASCII코드값
	실패시	EOF
설 명	fputc()함수와 같은 기능을 수행하며 사용법도 같다.	

◑ **putchar() 함수**

기 능	문자 하나를 stdout(표준출력)에 출력한다.	
헤 더	#include <stdio.h>	
선 언	int fputchar(int ch);	
형식인자	ch	출력할 문자의 ASCII코드 값
리 턴	성공시	출력한 문자의 ASCII코드값
	실패시	EOF
설 명	이 함수는 화면에 인수로 주어진 아스키 코드값에 해당하는 문자나 인수로 주어진 문자를 출력한다. 이 함수는「stdio.h」헤더파일에 매크로 함수로 정의되어 있다. #define putchar(c) putc((c), stdout) 즉 putchar()함수를 실행하면 putc()함수가 실행되며 이때 문자를 출력할 FILE 스트림은 stdout(표준출력)이 된다.	

▶ 아스키 문자의 세 부류

아스키 코드표를 보면 Text 모드에서 출력할 수 있는 256개의 아스키 문자들이 나타나 있다. 아스키 문자란 컴퓨터에서 문자를 표현하고자 정의한 것들인다 영문자(대소문자), 숫자, 아리비아 숫자와 특수문자들로 구성되어 있다.

1. 제어문자 (아스키 코드값 0 ~ 31) : 프린터와 같은 장치를 제어하는 기능
2. 일반문자 (아스키 코드값 32 ~ 127) : 아라비아 숫자, 알파벳 문자등을 나타냄
3. 확장 아스키 문자 (아스키 코드값 128 ~ 255) : 서유럽문자, 그래픽 문자, 선문자, 수학문자등

아스키 문자를 사용하는데 있어서 발생할 수 있는 문제점이 바로 키보드에 표현되어 있지 않는 확장아스키 문자들을 화면에 출력하고자 할 때 이다. 영문자나 아라비아 숫자 그리고 일부 특수문자를 키보드에 해당하는 키가 있어서 그 키를 사용하여 쉽게 출력할 수 있지만 확장아스키 문자는 해당하는 키가 없다.
이처럼 키보드에 없는 확장아스키 문자를 출력할 때 사용하는것이 아스키 코드값이다.

▶ 문자열출력함수

헤더	함수원형
stdio.h	int puts(const char *s);
stdio.h	int fputs(const char *s, FILE *fp);

◑ puts() 함수

기 능	문자열을 stdout(표준출력)에 출력한다.	
헤 더	#include <stdio.h>	
선 언	int puts(const char *s);	
형식인자	s	출력할 문자열을 가리키는 포인터
리 턴	성공시	음이 아닌 값을 반환
	실패시	매크로 EOF를 반환
설 명	표준출력이란 화면을 말한다. 곧 이 함수는 화면에 문자열 s를 출력하는 함수이다. puts()함수는 문자열 s를 출력하고 난 후 끝에 「LF」(개행)과 「CR」(복귀)을 덧붙이기 때문에, 커서의 위치가 다음 행 처음열로 이동하게 된다.	

◑ **fputs() 함수**

기 능	문자열을 file에 저장한다.	
헤 더	#include <stdio.h>	
선 언	int fputs(const char *s, FILE *fp);	
형식인자	s	출력할 문자열을 가리키는 포인터
	fp	출력할 파일을 가리키는 포인터
리 턴	성공시	문자열 s의 마지막 문자의 ASCII코드값
	실패시	매크로 EOF를 반환
설 명	문자열 s를 「*fp」가 가리키는 파일에 출력한다. 이 때 문자열의 끝을 나타내는 「'\0'」(널문자)는 출력하지 않으며 puts()함수처럼 「LF」(개행)과 「CR」(복귀)를 덧붙이지 않는다. 즉 문자열을 구성하는 순수한 문자들만을 출력한다.	

3. 문자열관련함수

문자열과 관련된 함수「string.h」에 선언되어 있으며 함수들은 크게 두가지로 분류할 수 있다.

▶「string.h」에 선언된 함수들의 분류

◑ str... : 함수명의 접두어가「str」인 함수들

str은「string」의 약자로써, 말 그대로 문자열에 관련된 함수들이다. 문자열의 특징인「'\0'」(널문자)를 이용하여 작업을 한다.

예) strcpy() 함수의 경우

문자열은「'\0'」(널문자)까지 복사된다.

◑ mem... : 함수명의 접두어가「mem」인 함수들

mem은「memory」의 약자로써, 컴퓨터 메모리에 관련된 함수들이다. 문자열의 특징인「'\0'」문자를 이용하지 않고 작업을 한다.

예) memcpy() 함수의 경우

복사되는 크기를 함수 인수로써 지정하여 복사한다.「'\0'」(널문자)와는 상관없이 복사된다.

▶ 복사함수

헤더	함수원형
string.h	char *strcpy(char *dest, const char *src);
string.h	char *stpcpy(char *dest, const char *src);
string.h	char *strncpy(char *dest, const char *src, size_t maxlen);
string.h	char *strdup(const char *s);
string.h	void *memcpy(void *dest, const void *src, size_t n);
string.h	void *memmove(void *dest, const void *src, size_t n);

◑ strcpy() 함수

기 능	src가 가리키는 문자열을 dest에 복사한다.
헤 더	#include <string.h>
선 언	char *strcpy(char *dest, const char *src);

형식인자	dest	복사 대상 문자열을 가리키는 포인터
	src	복사 원본 문자열을 가리키는 포인터
리 턴	문자열 dest를 가리키는 포인터	
설 명	이 함수를 실행하면 문자열 dest에 문자열 src가 복사되며, 리턴값 역시 문자열 src가 복사된 dest를 가리키는 포인터이다. 그러므로 이 함수를 실행하고나면 문자열 src와 dest, 그리고 리턴값은 모두 같은 문자열을 나타낸다. 이 함수를 사용함에 있어서 주의할 점은 dest의 기억공간의 크기가 src의 기억공간의 크기보다 작을 경우 기억공간의 경계범위를 벗어나 다른 기억장소를 침범하는 문제를 발생시킨다. 그러므로 되도록 이 함수보다는 복사할 문자열의 길이를 제한할 수 있는 strncpy()함수를 사용하도록 하는것이 좋다.	

◑ **stpcpy() 함수**

기 능	src가 가리키는 문자열을 dest에 복사한다.	
헤 더	#include <string.h>	
선 언	char *stpcpy(char *dest, const char *src);	
형식인자	dest	복사 대상 문자열을 가리키는 포인터
	src	복사 원본 문자열을 가리키는 포인터
리 턴	복사된 문자열의 마지막 문자 다음의 주소를 가리키는 포인터	
설 명	이 함수를 실행하면 문자열 dest에 문자열 src가 복사된다. strcpy()함수와의 차이점이 있다면 strcpy()함수가 문자열을 복사한 후, 복사된 문자열의 시작주소값을 반환하는 반면 stpcpy()함수는 문자열을 복사한 후, 복사된 문자열의 마지막 문자 다음 주소를 반환한다.	

◑ **strncpy() 함수**

기 능	src가 가리키는 문자열을 dest에 복사한다.	
헤 더	#include <string.h>	
선 언	char *strncpy(char *dest, const char *src, size_t manlen);	
형식인자	dest	복사 대상 문자열을 가리키는 포인터

	src	복사 원본 문자열을 가리키는 포인터
	maxlen	복사할 문자수
리 턴	문자열 dest를 가리키는 포인터	
설 명	이 함수는 strcpy()함수와 동일한 역할을 수행하나 차이점은 strcpy() 함수와는 달리 세번째 인수로써 복사할 데이터의 크기를 지정할 수 있다. 그러므로 strcpy()함수가 가지는 문제인 복사할 공간의 영역을 벗어나는 문제점을 해결할 수 있는 장점이 있으나 이로인해 「'\0'」(널문자)가 복사되지 않을 수 도 있는 문제점이 있다.	

◑ **strdup() 함수**

기 능	인수로 주어진 문자열을 복사하기 위한 기억공간을 할당하고, 문자열을 복사한다.	
헤 더	#include <string.h>	
선 언	char *strdup(const char *s);	
형식인자	s	복사할 문자열
리 턴	확보된 문자열 메모리의 첫 주소를 반환한다.	
설 명	이 함수는 인수로 주어진 문자열을 저장하기 위한 기억공간을 할당하고 문자열을 복사한 후, 할당된 기억공간의 시작주소를 반환한다. strcpy(), stpcpy(), strncpy()함수등이 문자열을 복사하기 위해 미리 할당된 기억공간을 사용하는 반면 이 함수는 자동으로 기억공간을 할당하고 문자열을 복사한 후, 복사된 문자열의 시작주소를 반환한다.	

◑ **memcpy() 함수**

기 능	메모리 영역을 복사한다.	
헤 더	#include <string.h>	
선 언	void *memcpy(void *dest, const void *src, size_t n);	
형식인자	dest	복사될 메모리의 포인터
	src	복사할 메모리의 포인터
	n	복사할 바이트 수
리 턴	성공시	데이터가 복사된 기억공간의 시작주소
	실패시	NULL

설 명	이 함수는 첫번째 인수로 주어진 기억공간에 두번째 인수로 주어진 기억공간의 데이터를 세번째 인수로 주어진 바이트 수 만큼 복사한다.

◑ **memmove() 함수**

기 능	메모리 영역을 복사한다.	
헤 더	#include <string.h>	
선 언	void *memmove(void *dest, const void *src, size_t n);	
형식인자	dest	복사될 메모리의 포인터
	src	복사할 메모리의 포인터
	n	복사할 바이트 수
리 턴	성공시	데이터가 복사된 기억공간의 시작주소
	실패시	NULL
설 명	이 함수는 memcpy()함수처럼 메모리 영역을 복사한다. memcpy()함수와 다른점은 하나의 포인터에 대해서 동일한 영역내에서 복사가 가능하다.	

▶ **비교함수**

헤더	함수원형
string.h	int strcmp(const char *s1, const char *s2);
string.h	int stricmp(const char *s1, const char *s2);
string.h	int strncmp(const char *s1, const char *s2, size_t maxlen);
string.h	int strincmp(const char *s1, const char *s2, size_t maxlen);
string.h	int memcmp(const void *s1, const void *s2, size_t n);

◑ **strcmp() 함수**

기 능	문자열 s1과 문자열 s2를 비교한다.	
헤 더	#include <string.h>	
선 언	int strcmp(const char *s1, const char *s2);	
형식인자	s1	비교할 원본 문자열을 가리키는 포인터

	s2	비교할 대상 문자열을 가리키는 포인터
리 턴	0	s1 == s2
	양의 수	s1 > s2
	음의 수	s1 < s2
설 명	s1, s2의 첫번째 문자부터 비교를 시작하여, 비교하는 문자가 서로 같지 않거나 문자열의 끝을 만나면 함수를 중단한다. 검사할 때 s1의 문자가 s2의 문자보다 크면 양수값을 반환하고, s1의 값이 s2보다 작으면 음수의 값을 반환한다. 여기서 문자의 값이 큰가 작은가는 ASCII코드값 또는 UNICODE값으로 비교하였을 때, 큰가 작은가를 의미한다. 그러므로 영문자 대문자는 항상 영문자 소문자보다 크다고 볼 수 있다.	

◑ **stricmp() 함수**

기 능	문자열 s1과 문자열 s2를 대소문자 구분없이 비교한다.	
헤 더	#include <string.h>	
선 언	int stricmp(const char *s1, const char *s2);	
형식인자	s1	비교할 원본 문자열을 가리키는 포인터
	s2	비교할 대상 문자열을 가리키는 포인터
리 턴	0	s1 == s2
	양의 수	s1 > s2
	음의 수	s1 < s2
설 명	이 함수는 strcmp()함수와 동일한 작업을 수행하지만 strcmp()함수가 대소문자를 구분하는 반면 이 함수는 대소문자를 구분하지 않고 비교한다. 이 함수와 동일한 역할을 수행하는 함수로는 strcasecmp()함수가 있으며 사용법도 동일하다.	

◑ **strncmp() 함수**

기 능	문자열 s1과 문자열 s2를 비교한다.	
헤 더	#include <string.h>	
선 언	int stricmp(const char *s1, const char *s2, size_t maxlen);	
형식인자	s1	비교할 원본 문자열을 가리키는 포인터

	s2	비교할 대상 문자열을 가리키는 포인터
	maxlen	비교할 문자열의 길이
리 턴	0	s1 == s2
	양의 수	s1 > s2
	음의 수	s1 < s2
설 명	이 함수는 strcmp()함수와 동일한 작업을 수행하지만 strcmp()함수와는 달리 비교할 문자의 수를 지정할 수 있다.	

◑ **memcmp() 함수**

기 능	두 포인터에 대한 데이터 비교	
헤 더	#include <string.h>	
선 언	int memcmp(const void *s1, const void *s2, size_t n);	
형식인자	s1	비교 대상 메모리 포인터
	s2	비교할 메모리 포인터
	n	비교할 바이트 수
리 턴	0	s1 == s2
	양의 수	s1 > s2
	음의 수	s1 < s2
설 명	s1이 가리키는 기억공간의 n바이트 만큼의 데이터와 s2가 가리키는 기억공간의 n바이트 만큼의 데이터를 비교한다.	

▶ **연결함수**

헤더	함수원형
string.h	char *strcat(char *dest, const char *src);
string.h	char *strncat(char *dest, const char *src, size_t n);

◑ **strcat() 함수**

기 능	두개의 문자열을 연결한다.
헤 더	#include <string.h>

선 언	char *strcat(char *dest, const char *src);	
형식인자	dest	추가되는 문자열을 가리키는 포인터
	src	추가할 문자열을 가리키는 포인터
리 턴	변경된 문자열 dest를 가리키는 포인터	
설 명	이 함수는 두개의 문자열 포인터를 인수로 받아서 하나의 문자열로 만든다. 이때 합쳐진 문자열은 첫번째 인수로 주어진 포인터가 가리키는 배열에 저장된다. 그러므로 첫번째 인수가 가리키는 포인터의 배열은 합쳐진 문자열의 크기보다 1개 이상 더 커야만 한다.	

◑ **strncat() 함수**

기 능	두개의 문자열을 연결한다.	
헤 더	#include <string.h>	
선 언	char *strcat(char *dest, const char *src, size_t n);	
형식인자	dest	추가되는 문자열을 가리키는 포인터
	src	추가할 문자열을 가리키는 포인터
리 턴	변경된 문자열 dest를 가리키는 포인터	
설 명	이 함수는 strcat()함수와 동일하지만 strcat()함수와는 달리 연결할 문자열의 갯수를 지정할 수 가 있다.	

▶ **검색함수**

헤더	함수원형
string.h	char *strchr(const char *s, int c);
string.h	char *strrchr(const char *s, int c);
string.h	char *strstr(const char *s1, const char *s2);
string.h	char *strpbrk(const char *s1, const char *s2);
string.h	void *memchr(const void *s1, int c, size_t n);

◑ **strchr() 함수**

기 능	문자열 s에 문자 c가 있는지 앞에서 부터 검색한다.

헤 더	#include <string.h>	
선 언	char *strchr(const char *s, int c);	
형식인자	s	검색대상 문자열을 가리키는 포인터
	c	검색할 문자
리 턴	포인터	문자 c의 위치를 가리키는 포인터
	NULL	문자열 s에 문자 c가 없는 경우
설 명	문자열 s에서 문자 c가 있는지 앞에서부터 검색하여, 찾아낼 경우 문자 c의 위치를 가리키는 포인터를 반환한다. 즉 리턴값은 문자열 s에서 찾아낸 문자 c의 뒷부분을 구성하는 문자열이다.	

◑ **strrchr() 함수**

기 능	문자열 s에 문자 c가 있는지 앞에서 부터 검색한다.	
헤 더	#include <string.h>	
선 언	char *strrchr(const char *s, int c, size_t n);	
형식인자	s	검색대상 문자열을 가리키는 포인터
	c	검색할 문자
리 턴	포인터	문자 c의 위치를 가리키는 포인터
	NULL	문자열 s에 문자 c가 없는 경우
설 명	문자열 s에서 문자 c가 있는지 앞에서부터 검색하여, 찾아낼 경우 문자 c의 위치를 가리키는 포인터를 반환한다. 즉 리턴값은 문자열 s에서 찾아낸 문자 c의 뒷부분을 구성하는 문자열이다.	

◑ **strstr() 함수**

기 능	문자열 s1에 문자열 s2가 있는지를 검색한다.	
헤 더	#include <string.h>	
선 언	char *strstr(const char *s1, const char *s2);	
형식인자	s1	검색대상 문자열을 가리키는 포인터
	s2	검색할 문자열
리 턴	포인터	s1내의 문자열에서 s2의 위치를 가리키는 주소
	NULL	문자열 s1에 문자열 s2가 없는 경우

설 명	이 함수는 문자열에서 임의의 문자열이 시작하는 위치를 구한다.

◑ **strpbrk() 함수**

기 능	문자열에서 지정된 문자들이 있는 위치를 반환한다.	
헤 더	#include <string.h>	
선 언	char *strpbrk(const char *s1, const char *s2);	
형식인자	s1	검색대상 문자열을 가리키는 포인터
	s2	검색에 사용되는 문자들의 모임
리 턴	포인터	문자열 s1에서 s2에 포함되어 있는 문자의 최초 위치의 주소
	NULL	문자열 s1에 s2에 포함된 문자가 없는 경우
설 명	이 함수는 첫번째 인수로 주어진 s1에서 두번째 인수로 주어진 s2에 저장된 문자들을 검색하여 최초로 검색된 위치의 주소값을 반환하게 된다.	

◑ **memchr() 함수**

기 능	문자열 s에 문자 c가 있는지 뒤에서 부터 검색한다.	
헤 더	#include <string.h>	
선 언	void *memchr(const void *s, int c, size_t);	
형식인자	s	검색대상 포인터
	c	검색값
	n	검색할 바이트 수
리 턴	포인터	s상에서 검색된 최초 c의 주소
	NULL	검색된 값이 없을 경우
설 명	이 함수는 첫번째 인수로 주어진 포인터가 가리키는 기억공간으로 부터 두번째 인수로 주어진 값을 검색한다. 이때 검색은 첫번째 인자로 주어진 포인터가 가리키는 주소로 부터 세번째 인수로 주어진 바이트 수 만큼 검색한다. [예제] 문자열에서 특정문자 검색하기 #include <stdio.h> #include <stdio_ext.h>	

```c
#include <string.h>

int main()
{
        char *s = "Hello C Programming!!!";
        char *sp;

        sp = memchr(s, 'C', sizeof(s));
        printf("%s\n", sp);

        __fpurge(stdin);
        getchar();
        return 0;
}
```

[예제] 배열에서 특정값 검색하기

```c
#include <stdio.h>
#include <stdio_ext.h>
#include <string.h>

int main()
{
        int arr[] = {100, 200, 300, 400, 500};
        int *ap;

        ap = (int *)memchr(arr, 300, sizeof(arr));
        printf("ap의 주소 : %p\n", ap);
        printf("*ap의 값 : %d\n", *ap);

        __fpurge(stdin);
        getchar();
        return 0;
}
```

▶ **문자수검색함수**

헤더	함수원형
string.h	size_t strlen(const char *s);
string.h	size_t strspn(const char *s1, const char *s2);
string.h	size_t strcspn(const char *s1, const char *s2);

◑ **strlen() 함수**

기 능	문자열 s의 길이를 구한다.

헤 더	#include <string.h>	
선 언	int strlen(const char *s);	
형식인자	s	문자열을 가리키는 포인터
리 턴	문자열 s의 길이(바이트 수)를 반환한다.	
설 명	인수로 주어진 포인터가 가리키는 곳의 문자열의 길이(바이트 수)를 반환한다.	

◑ **strspn() 함수**

기 능	특정 문자로 구성된 문자열의 길이를 구한다.	
헤 더	#include <string.h>	
선 언	int strspn(const char *s1, const char *s2);	
형식인자	s1	검색대상 문자열
	s2	검색에 사용되는 문자들의 모임
리 턴	s2의 문자들만을 포함하고 있는 s1의 처음 부분의 길이를 구한다	
설 명	s2의 문자들만을 포함하고 있는 s1 의 처음 부분의 길이를 구한다. 예를들어 s2 에 "12346"이 들어있고 s1 에는 "321ab"가 들어 있다면 s1 의 맨 처음 부터 문자들을 비교하는데, s2 에 들어 있는 문자들 만을 포함한 곳 까지의 길이, 이 경우에는 3,2,1 이 해당하므로 3 이 리턴된다.	

◑ **strcspn() 함수**

기 능	문자열 s의 길이를 구한다.	
헤 더	#include <string.h>	
선 언	int strcspn(const char *s1, const char *s2);	
형식인자	s1	검색대상 문자열
	s2	검색할 문자열
리 턴	s1에서 s2에 포함된 문자 중 첫번째로 일치하는 문자에 도달하기 까지 읽어들인 문자의 개수	
설 명	문자열에서 다른 문자열에 포함되어 있는 문자들을 검색하되, 첫번째로 일치하는 문자에 도달하기 까지 읽어들인 문자의 개수를 리턴하게 된다.	

	예를 들어 s1 에 "Hello, World" 가 들어 있고, s2 에 "aeiou" 가 들어 있다면 s2 에 들어 있는 문자들, 즉 a,e,i,o,u 를 s1 에서 검색하여 첫번째로 일치하는 것, 즉 'e' 까지 읽어들이는데 읽은 문자의 수, 즉 1 을 리턴하게 된다.

▶ **초기화함수**

헤더	함수원형
string.h	void *memset(void *s, int c, size_t n);

◑ **memset() 함수**

기 능	메모리를 특정값으로 초기화 한다.	
헤 더	#include <string.h>	
선 언	void *memset(void *s, int c, size_t n);	
형식인자	s	초기화할 기억공간을 가리키는 포인터
	c	초기화에 사용되는 값
	n	초기화할 바이트 수
리 턴	성공시	초기화된 기억공간의 시작주소
	실패시	NULL
설 명	이 함수는 malloc()이나 calloc()함수에 의해 동적으로 할당된 기억공간을 초기화 할때 사용하는 함수이다. calloc()함수에 의해 할당된 기억공간은 자동으로 초기화 되는 반면 malloc()함수에 의해 할당된 기억공간은 자동으로 초기화가 이뤄지지 않는다. 이때 이 함수를 이용하여 초기화를 할 수 있다.	

▶ **문자열 입력함수**

헤더	함수원형
string.h	char *gets(char *buffer);
string.h	char *fgets(char *buffer, int n, FILE *fp);

◑ **gets() 함수**

기 능	문자열을 stdin(표준입력)으로부터 입력받는다.

헤 더	#include <stdio.h>	
선 언	char *gets(char *buffer);	
형식인자	buffer	입력받은 문자열을 저장할 포인터
리 턴	성공시	문자열을 가리키는 포인터
	실패시	매크로 NULL을 반환
설 명	키보드로부터 문자열을 입력받는 함수이다. 키보드로 문자열을 입력한 후, 「Enter」키를 누르면 입력이 종료되며 「Enter」키의 값은 「'\0'」(널문자)로 변환되어 저장된다. 따라서 buffer는 적어도 입력받고자 하는 문자의 갯수 + 1개 만큼의 크기이어야만 한다. 만약 입력받은 문자열을 저장할 buffer의 길이가 입력된 문자열의 갯수 + 1개 보다 작은 경우 다른 변수의 영역을 침범할 우려가 있다. 또한 scanf()함수의 경우 문자열을 입력받을 때 공백문자는 입력받을 수 없지만 gets()함수는 공백문자도 입력받을 수 있다. 또한 gets()함수는 「Ctrl + z」가 입력되면 NULL을 리턴한다.		

◑ **fgets() 함수**

기 능	문자열을 file로부터 입력받는다.	
헤 더	#include <stdio.h>	
선 언	char *fgets(char *buffer, int n, FILE *fp);	
형식인자	buffer	입력받은 문자열을 저장할 포인터
	n	읽어들일 문자열의 길이
	fp	읽어들일 파일을 가리키는 포인터
리 턴	성공시	s를 가리키는 포인터
	실패시	매크로 NULL을 반환
설 명	지정한 파일로부터 n-1개 만큼 문자들을 읽어들여 s에 저장한다. s의 끝에는 「'\0'」(널문자)를 추가한다. 이때 읽어들인 문자가 만약 파일의 끝인 EOF라면 입력을 중단하며 EOF는 문자열 s에 「'\0'」(널문자)로 변환되어 저장된다. fgets()함수는 입력받을 문자열의 길이를 지정할 수 있으나 지정한 문자열의 갯수에 도달하지 않더라도 「'\n'」(개행문자)를 만나면 입력을 중단한다. 「'\n'」(개행문자)는 「'\0'」(널문자)로 변환되지 않고 그대로 저장되며, 그 대신 s의 끝에 「'\0'」(널문자)를 추가한다.		

▶ **문자열출력함수**

헤더	함수원형
string.h	int puts(const char *s);
string.h	int fputs(const char *s, FILE *fp);

◑ **puts() 함수**

기 능	문자열을 stdout(표준출력)에 출력한다.	
헤 더	#include <stdio.h>	
선 언	int puts(const char *s);	
형식인자	s	출력할 문자열을 가리키는 포인터
리 턴	성공시	음이 아닌 값을 반환
	실패시	매크로 EOF를 반환
설 명	표준출력이란 화면을 말한다. 곧 이 함수는 화면에 문자열 s를 출력하는 함수이다. puts()함수는 문자열 s를 출력하고 난 후 끝에 「LF」(개행)과 「CR」(복귀)을 덧붙이기 때문에, 커서의 위치가 다음 행 처음열로 이동하게 된다.	

◑ **fputs() 함수**

기 능	문자열을 file에 저장한다.	
헤 더	#include <stdio.h>	
선 언	int fputs(const char *s, FILE *fp);	
형식인자	s	출력할 문자열을 가리키는 포인터
	fp	출력할 파일을 가리키는 포인터
리 턴	성공시	문자열 s의 마지막 문자의 ASCII코드값
	실패시	매크로 EOF를 반환
설 명	문자열 s를 「*fp」가 가리키는 파일에 출력한다. 이 때 문자열의 끝을 나타내는 「'\0'」(널문자)는 출력하지 않으며 puts()함수처럼 「LF」(개행)과 「CR」(복귀)를 덧붙이지 않는다. 즉 문자열을 구성하는 순수한 문자들만을 출력한다.	

4. 데이터 변환 함수

▶ 데이터 변환 함수

헤더	함수원형
stdlib.h	int atoi(const char *str);
stdlib.h	long atol(const char *str);
stdlib.h	int atof(const char *str);
stdlib.h	long strtol(const char *str, char **endptr, int base);
stdlib.h	unsinged long strtoul(const char *str, char **endptr, int base);
stdlib.h	float strtof(const char *str, char **endptr);
stdlib.h	double strtod(const char *str, char **endptr);

◑ atoi() 함수

기 능	문자열을 10진수 int형 정수로 변환한다.	
헤 더	#include <stdlib.h>	
선 언	int atoi(const char *str);	
형식인자	str	정수로 변환할 문자열을 가리키는 포인터
리 턴	10진수로 변환된 정수값	
설 명	이 함수는 인자로 주어진 문자열을 10진수 정수로 변환한다. 만약 변환하고자 하는 문자열에 숫자가 아닌 문자가 포함되어 있는 경우 숫자문자까지만 10진수로 변환되며 첫번째 문자가 첫번째 문자부터 변환이 실패할 경우에는 0값을 반환한다. 숫자문자로 시작하는 경우 「134234」,「-134234」,「+134234」,「 134234」 숫자문자로 시작하지 않는 경우 「a34234」,「- 134234」 숫자문자로 시작하며 숫자문자 앞에 공백문자가 있는 경우에는 공백문자는 자동으로 제거되어 변환된다.	

	만약 변환하고자 하는 문자열이 int형으로 표현할 수 있는 범위를 넘어서는 경우 오버 플로우 또는 언더 플로우 된 값이 저장되게 된다. 예를들어 변환하고자 하는 문자열이 "-2147483649" 라면 언더플우가 된 값인 2147483647값이 반환된다.

◑ **atol() 함수**

기 능	문자열을 10진수 int형 정수로 변환한다.	
헤 더	#include <stdlib.h>	
선 언	int atol(const char *str);	
형식인자	str	정수로 변환할 문자열을 가리키는 포인터
리 턴	10진수로 변환된 정수값	
설 명	이 함수는 인자로 주어진 문자열을 10진수 long형 정수로 변환하며 사용법은 atoi()함수와 동일하다.	

◑ **atof() 함수**

기 능	문자열을 실수형 데이터로 변환한다.	
헤 더	#include <stdlib.h>	
선 언	double atof(const char *str);	
형식인자	str	실수로 변환할 문자열을 가리키는 포인터
리 턴	10진수로 변환된 double형 실수 값	
설 명	이 함수는 인수로 주어진 문자열을 10진수 double형 실수로 변환한다. 사용법은 atoi()함수와 동일하다. 만약 문자열에 소수점을 나타내는 「.」이 여러개가 있는 경우, 첫번째 「.」이 소수점이 되며 두번째 「.」는 실수로 변환할 수 없는 문자로 인식한다.	

◑ **strtol() 함수**

기 능	문자열을 long형 정수로 변환한다.	
헤 더	#include <stdlib.h>	
선 언	long strtol(const char restrict *str, char **endptr, int base);	
형식인자	str	정수로 변환할 문자열

	endptr	숫자로 변경하지 못한 문자의 주소값을 저장할 포인터
	base	문자열이 표현하고 있는 진법 (2 ~ 32)
리 턴	10진수 long형 정수로 변환된 값	
설 명	strtol()함수는 문자열을 long형 숫자로 변환한다. 이 함수는 atoi() 함수나 atol()함수와는 달리 변환하고자 하는 진수를 선택할 수 있으며 변환 대상 문자열에서 숫자로 변환하지 못한 문자의 주소값을 구할 수 있다. 이 함수는 문자열이 변환할 수 없는 문자로 시작하는 경우 0값을 반환한다. 이 경우 변환하지 못하여 0값을 반환한 것인지 아니면 변환 된 결과가 0값인지는 두번째 인수로 주어진 포인터의 위치를 이용하여 구분할 수 있다. 또한 변환하고자 하는 문자열이 long형으로 표현할 수 있는 범위 밖의 수 인 경우 long형으로 표현할 수 있는 최소값 또는 최대값이 반환된다. 이것 또한 문자열이 long형으로 표현할 수 있는 최소값 또는 최대값이어서 반환된 값 인지 아니면 범위를 벗어났기때문에 최소값 또는 최대값인지를 구분할 수 없게 되는 데, 이 때에는 errno값을 참조하여 구분할 수가 있다. 두번째 인수는 변환하고자 하는 문자열에서 숫자로 변환하지 못한 문자의 주소를 저장하기 위한 인수이다. 만약 이 두번째 인수의 값으로 NULL을 넘겨주면 변환하지 못한 문자의 주소값을 구하지 않게 된다. errno를 사용하기 위해서는 errno.h 헤더를 include시켜야 하며 strtol()함수를 호출하기전에 반드시 errno값을 0으로 초기화 하여야 한다.	

◑ **strtoul() 함수**

기 능	문자열을 unsigned long형 정수로 변환한다.	
헤 더	#include <stdlib.h>	
선 언	long strtoul(const char restrict *str, char **endptr, int base);	
형식인자	str	정수로 변환할 문자열
	endptr	숫자로 변경하지 못한 문자의 주소값을 저장할 포인터

	base	문자열이 표현하고 있는 진법 (2 ~ 32)
리 턴	10진수 unsigned long형 정수로 변환된 값	
설 명	strtol()함수와 동일	

◑ **strtof() 함수**

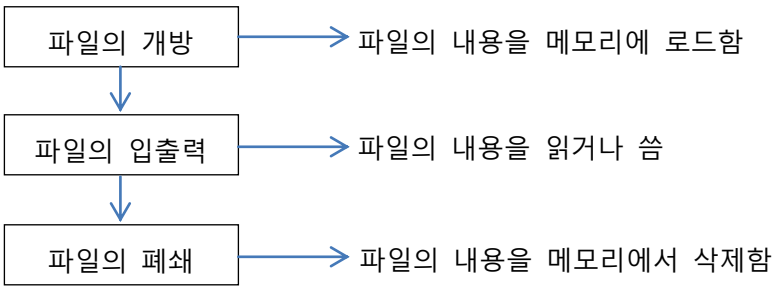
기 능	문자열을 float형 실수로 변환한다.	
헤 더	#include <stdlib.h>	
선 언	float strtof(const char *str, char **endptr);	
형식인자	str	실수로 변환할 문자열
	endptr	숫자로 변경하지 못한 문자의 주소값을 저장할 포인터
리 턴	변환된 float형 실수 값	
설 명	이 함수는 인수로 주어진 문자열을 float형 실수로 변환한다. 두번째 인수는 변환문자열에서 실수로 변환하지 못한 문자의 주소값을 저장하는 인수이다. 이 두번째 인수의 값으로 NULL을 지정하면 변환하지 못한 문자의 주소값을 구하지 않는다.	

◑ **strtod() 함수**

기 능	문자열을 double형 실수로 변환한다.	
헤 더	#include <stdlib.h>	
선 언	long strtod(const char *str, char **endptr);	
형식인자	str	실수로 변환할 문자열
	endptr	숫자로 변경하지 못한 문자의 주소값을 저장할 포인터
리 턴	변환된 double형 실수 값	
설 명	이 함수는 인수로 주어진 문자열을 double형 실수로 변환하며 사용법은 strtof()함수와 동일하다.	

5. 파일관련함수

▶ 파일의 입출력 작업



▶ 파일의 개방

파일의 개방은 파일 입출력 작업에 앞서 파일의 내용을 디스크에서 메모리에 옮겨 놓는 작업이다. 파일은 아래의 두가지 형태로 개방할 수 있다.

- ◑ 텍스트모드
- ◑ 이진(바이너리)모드

파일을 쓸 때 라인넘김은 시스템 마다 다르게 표현된다. 예를들어 윈도우즈 시스템에서의 개행은 「"\r\n"」(CR + LF)인 반면 리눅스와 같은 POSIX시스템에서의 개행문자는 「'\n'」(LF)이다.

예를들어 윈도우즈 시스템에서 파일을 텍스트 모드로 개방하였을 경우 파일에 「'\n'」(LF)을 기록하면 라이브러리 함수에서는 자동으로 「"\r\n"」(CR + LF)로 변환하여 저장하게 된다. 또한 읽어들일 때 역시 「"\r\n"」(CR + LF)를 「'\n'」(LF)로 변환하여 읽어들이게 된다. 하지만 바이너리 모드로 개방을 하는 경우 파일 입출력시 개행문자에 대해 아무런 변형없이 파일의 내용을 그대로 입출력하게 된다.

▶ 파일개방함수

헤더	함수원형
stdio.h	FILE *fopen(const char *path, const char *mode);
fcntl.h	open(const char *path, int flags[, int mode]);
fcntl.h	create(const char *path, mode_t mode);
stdio.h	FILE *fdopen(int fd, const char *mode);

unistd.h	int dup(int fd);
unistd.h	int dup2(int fd, int num);

◑ **fopen() 함수**

기 능	파일을 개방하는 함수(고수준레벨오픈)	
헤 더	#include <stdio.h>	
선 언	FILE *fopen(const char *path, const char *mode);	
형식인자	path	개방할 파일의 경로
	mode	파일의 개방모드
리 턴	성공시	파일 포인터를 반환
	실패시	NULL을 반환

설 명

이 함수는 첫번째 인수에 지정된 경로의 파일을 개방하는 함수이다. 이때 두번째 인수로써 파일의 개방모드를 지정하며 모드의 지정은 다음과 같은 플래그를 이용한다.

"r"	read모드로 파일을 개방한다. 이 모드는 반드시 파일이 존재하여야만 한다.
"w"	write모드로 파일을 개방한다. 파일이 존재하지 않는 경우 파일을 생성한다.
"a"	append모드로 파일을 개방한다. 파일이 존재하지 않는 경우 파일을 생성한다. write모드와 다른점은 write모드는 파일의 데이터를 삭제한 후 새로 쓰기를 하는 반면 이 모드는 기존의 데이터 끝에서부터 기록하게 된다.
"r+"	읽기와 쓰기 모드로 파일을 개방한다.
"w+"	읽기와 쓰기 모드로 파일을 개방한다.
"rt"	텍스트파일 읽기 전용으로 개방한다.
"wt"	텍스트파일 쓰기 전용으로 개방한다.
"at"	텍스트파일 추가용으로 파일을 개방한다.
"rb"	바이터리파일 읽기 전용으로 파일을 개방한다.
"wb"	바이너리파일 쓰기 전용으로 파일을 개방한다.
"ab"	바이너리파일 추가용으로 파일을 개방한다.

◑ **open() 함수**

기 능	파일을 개방하는 함수(저수준레벨오픈)	
헤 더	#include <fcntl.h>	
선 언	int open(const char *path, int flags[, mode_t mode]);	
형식인자	path	개방할 파일의 경로
	flags	파일열기에 대한 옵션
	mode	O_CREAT 옵션 사용에 의해 파일이 생성될 때 파일에 지정되는 접근권한
리 턴	성공시	오픈된 파일의 디스크립터
	실패시	-1을 반환
설 명	이 함수는 파일을 개방하는 함수로써 파일을 오픈할 때 용도에 따라 읽기전용, 쓰기전용 또는 읽기와 쓰기가 모두 가능한 옵션을 지정하여 오픈한다.	

O_RDONLY	파일을 읽기 전용으로 개방한다.
O_WRONLY	파일을 쓰기 전용으로 개방한다.
O_RDWR	읽기/쓰기 모두 가능하도록 개방한다.
O_CREAT	해당 파일이 없으면 파일을 생성한다.
O_EXCL	O_CREAT 플래그에 의해 파일을 생성할 때 기존에 동일한 이름의 파일이 존재하는 경우 새로운 파일이 생성되지 않도록 한다.
O_TRUNC	기존의 파일내용을 모두 삭제한다.
O_APPEND	파일을 추가하여 쓰기가 가능하도록 개방한 후, 파일의 포인터를 끝에 위치시킨다.
O_SYNC	쓰기를 할 때, 실제 쓰기가 완료될 때 까지 대기한다. 즉 물리적으로 쓰기가 완료되어야 복귀하게 된다.

◑ **create() 함수**

기 능	파일을 생성한다.	
헤 더	#include <fcntl.h>	
선 언	int create(const char *path, int mode);	
형식인자	path	경로를 포함한 생성할 파일의 경로

	mode	생성할 파일의 접근권한
리 턴	성공시	생성한 파일의 디스크립터
	실패시	-1을 반환
설 명	이 함수는 파일을 생성하는 함수이다. 이 함수는 새로운 파일을 생성한 후 파일의 개방까지 수행하게 된다. 만약 생성하고자 하는 이름의 파일이 이미 존재하는 경우 파일의 내용을 모두 삭제하고 개방을 하게 된다. create()함수는 open()함수에 비해 간단하게 보이지만 create()함수보다는 되도록 open()함수를 이용하도록 하는것이 좋다. create()함수는 open()함수에 비해 개방옵션이 고정되어 있기때문이다. create()함수는 파일을 생성하고 개방할 때 쓰기만 가능하기 때문에 읽기가 가능하도록 하려면 파일을 폐쇄한 후 다시 개방하여야만 한다.	

◑ **fdopen() 함수**

기 능	파일 디스크립터로부터 파일 구조체 포인터를 구한다.	
헤 더	#include <stdio.h>	
선 언	FILE *fdopen(int fd, const char *mode);	
형식인자	fd	파일 구조체 포인터를 구할 파일 디스크립터
	mode	파일을 열기위한 옵션
리 턴	성공시	FILE 구조체 포인터
	실패시	NULL
설 명	이 함수는 open()함수에 의해 구해진 파일 디스크립터로부터 FILE 구조체 포인터를 구해낸다.	

◑ **dup() 함수**

기 능	파일 디스크립터의 복사본을 생성한다.	
헤 더	#include <unistd.h>	
선 언	int dup(int fd);	
형식인자	fd	복사할 파일 디스크립터
리 턴	성공시	복사하여 새로 생성된 파일 디스크립터 번호
	실패시	-1
설 명	이 함수는 open()함수에 의해 구해진 파일 디스크립터로부터 FILE 구조체 포인터를 구해낸다.	

◑ **dup2() 함수**

기 능	파일 디스크립터의 복사본을 생성한다.	
헤 더	#include <unistd.h>	
선 언	int dup(int fd, int num);	
형식인자	fd	복사할 파일 디스크립터
	number	새로 생성할 파일 디스크립터의 번호
리 턴	성공시	복사하여 새로 생성된 파일 디스크립터 번호
	실패시	-1
설 명	이 함수는 dup()함수와 동일하지만 dup()함수가 커널에서 자동으로 파일 디스크립터 번호를 지정하는 반면 dup2()함수는 프로그래머가 지정하는 번호 파일 디스크립터를 복사한다. 만약 지정한 디스크립터의 번호가 이미 사용중이라면 해당 파일을 닫은 후, 다시 지정한다.	

▶ **파일입출력함수**

헤더	함수원형
stdio.h	int fscanf(FILE *fp, const char *format, ...);
stdio.h	int vfscanf(FILE *fp, const char *format, va_list arglist);
stdio.h	int fgetc(FILE *fp);
stdio.h	int getc(FILE *fp);
stdio.h	char *fgets(char *buffer, int n, FILE *fp);
stdio.h	int fprintf(FILE *fp, const char *format, ...);
stdio.h	int vfprintf(FILE *fp, const char *format, va_list arglist);
stdio.h	int fputc(int ch, FILE *fp);
stdio.h	int putc(int c, FILE *fp);
stdio.h	int fputs(const char *str, FILE *fp);
stdio.h	size_t fwrite(void *ptr, size_t size, size_t n, FILE *fp);
stdio.h	size_t fread(void *ptr, size_t size, size_t n, file *fp);

unistd.h	ssize_t write(int fd, const void *buf, size_t n);
unistd.h	ssize_t read(int fd, void *buf, size_t n);

◑ **fscanf() 함수**

기 능	format 형태대로 파일로 부터 입력받는다.	
헤 더	#include <stdio.h>	
선 언	int fscanf(FILE *fp, const char * format, ...);	
형식인자	fp	데이터를 입력받을 파일을 가리키는 FILE 구조체 포인터
	format	입력받을 서식
	...	입력된 값을 저장할 인수들
리 턴	성공시	입력된 서식의 수
	문자열의 끝을 입력받은 경우 매크로 EOF를 반환함	
설명		

◑ **vfscanf() 함수**

기 능	format 형태대로 파일로부터 입력받는다.	
헤 더	#include <stdio.h>	
선 언	int vfscanf(FILE *fp, const char *format, va_list arglist);	
형식인자	fp	데이터를 입력받을 파일을 가리키는 FILE구조체 포인터
	format	입력받을 서식
	arglist	입력된 데이터를 저장할 가변개수 인수를 가리키는 포인터
리 턴	성공시	입력된 서식의 갯수
	문자열의 끝을 입력받은 경우 EOF를 반환 함	
설 명		

◑ **fgetc() 함수**

기 능	문자 하나를 파일로부터 입력받는다.	
헤 더	#include <stdio.h>	
선 언	int fgetc(FILE *fp);	
형식인자	fp	문자를 읽어들일 파일을 가리키는 포인터

리 턴	성공시	입력된 문자의 ASCII 코드값
	실패시	매크로 EOF를 반환
설 명	이 함수는 지정한 파일로부터 한 문자를 입력받고, 그 문자의 아스키 코드값을 반환한다.	

◑ **getc() 함수**

기 능	문자 하나를 파일로부터 입력받는다.	
헤 더	#include <stdio.h>	
선 언	int getc(FILE *fp);	
형식인자	fp	문자를 읽어들일 파일을 가리키는 포인터
리 턴	성공시	입력된 문자의 ASCII 코드값
	실패시	매크로 EOF를 반환
설 명	fgetc()함수와 같은 기능을 수행하며 사용법도 동일하다.	

◑ **fgets() 함수**

기 능	문자열을 file로부터 입력받는다.	
헤 더	#include <stdio.h>	
선 언	char *fgets(char *buffer, int n, FILE *fp);	
형식인자	buffer	입력받은 문자열을 저장할 포인터
	n	읽어들일 문자열의 길이
	fp	읽어들일 파일을 가리키는 포인터
리 턴	성공시	s를 가리키는 포인터
	실패시	매크로 NULL을 반환
설 명	지정한 파일로부터 n-1개 만큼 문자들을 읽어들여 s에 저장한다. s의 끝에는 「'\0'」(널문자)를 추가한다. 이때 읽어들인 문자가 만약 파일의 끝인 EOF라면 입력을 중단하며 EOF는 문자열 s에 「'\0'」(널문자)로 변환되어 저장된다. fgets()함수는 입력받을 문자열의 길이를 지정할 수 있으나 지정한 문자열의 갯수에 도달하지 않더라도 「'\n'」(개행문자)를 만나면 입력을 중단한다. 「'\n'」(개행문자)는 「'\0'」(널문자)로 변환되지 않고 그대로 저장되며, 그 대신 s의 끝에 「'\0'」(널문자)를 추가한다.	

◑ **fprintf() 함수**

기 능	format 형태대로 파일에 출력한다.	
헤 더	#include <stdio.h>	
선 언	int fprintf(const char *format, ...);	
형식인자	fp	출력할 파일을 가리키는 FILE 구조체 포인터
	format	출력할 때 의 서식
	...	인수들
리 턴	성공시	출력한 문자들의 바이트 수
	실패시	EOF
설 명	파일에 내용을 출력하기 위해선 단지 파일 출력 함수인 fprintf() 함수만을 사용하는 것이 아니라, fopen()이나 fclose() 함수와 같이 파일을 열거나 닫는 함수를 같이 사용하여야 한다.	

◑ **vfprintf() 함수**

기 능	format 형태대로 문자열을 파일에 저장한다.	
헤 더	#include <stdio.h>	
선 언	int vfprintf(FILE *fp, const char *format, va_list arglist);	
형식인자	fp	출력할 파일을 가리키는 FILE구조체 포인터
	format	출력할 때 의 서식
	arglist	가변개수 인수를 가리키는 포인터
리 턴	성공시	출력한 문자들의 바이트 수
	실패시	EOF
설 명	vfprintf() 함수의 사용법은 fprintf() 함수와 비슷하다. 차이점이라면 출력의 결과가 stdout이 아닌 첫번째 파라미터로 주어진 포인터가 가리키는 파일에 저장된다는 것이다. 또한 fprintf()함수는 「...」을 사용하여 가변 개수 인수들을 하나 하나 전달받는 방식이지만 vfprintf()함수는 가변인수들을 가리키는 포인터인 arglist를 사용한다.	

◑ **fputc() 함수**

기 능	문자 하나를 file에 출력한다.

헤 더	#include <stdio.h>	
선 언	int fputc(int ch, FILE *fp);	
형식인자	ch	출력할 문자의 ASCII코드 값
	file	문자를 출력할 파일을 가리키는 FILE구조체 포인터
리 턴	성공시	출력한 문자의 ASCII코드값
	실패시	EOF
설 명	출력할 문자를 나타내는 인수 「ch」에 아스키 코드값이나 문자상수를 지정하면 아스키 코드값에 해당하는 문자나 해당 문자가 「fp」가 가리키는 파일에 출력된다.	

◑ **fputs() 함수**

기 능	문자열을 file에 저장한다.	
헤 더	#include <stdio.h>	
선 언	int fputs(const char *s, FILE *fp);	
형식인자	s	출력할 문자열을 가리키는 포인터
	fp	출력할 파일을 가리키는 포인터
리 턴	성공시	문자열 s의 마지막 문자의 ASCII코드값
	실패시	매크로 EOF를 반환
설 명	문자열 s를 「*fp」가 가리키는 파일에 출력한다. 이 때 문자열의 끝을 나타내는 「'\0'」(널문자)는 출력하지 않으며 puts()함수처럼 「LF」(개행)과 「CR」(복귀)를 덧붙이지 않는다. 즉 문자열을 구성하는 순수한 문자들만을 출력한다.	

◑ **fread() 함수**

기 능	fopen()함수에 의해 열려진 파일로부터 읽기를 한다.	
헤 더	#include <stdio.h>	
선 언	size_t fread(void *buf, size_t size, size_t n, FILE *fp);	
형식인자	buf	파일의 내용을 읽어들여 저장할 포인터
	size	데이터 하나의 크기
	n	읽어들일 데이터 수

	fp	파일 구조체 포인터
리 턴	성공시	읽어들인 데이터의 수
	실패시	-1
설 명		

◑ **fwrite() 함수**

기 능	fopen()함수에 의해 열려진 파일에 쓰기를 한다.	
헤 더	#include <stdio.h>	
선 언	size_t fwrite(const void *buf, size_t size, size_t n, FILE *fp);	
형식인자	buf	출력할 내용을 가리키는 포인터
	size	데이터 하나의 크기
	n	쓰기를 할 데이터 개수
	fp	파일 구조체 포인터
리 턴	성공시	출력한 데이터 수 (바이트 수)
		-1
설 명		

◑ **read() 함수**

기 능	open()함수로 열기를 한 파일로부터 데이터를 읽어들인다.	
헤 더	#include <unistd.h>	
선 언	ssize_t read(int fd, void *buf, size_t n);	
형식인자	fd	파일 디스크립터
	buf	읽어들인 데이터를 저장할 공간을 가리키는 포인터
	n	읽어들일 문자 수(바이트 수)
리 턴	성공시	읽어들인 문자의 수(바이트 수)
	실패시	-1
설 명	표준 출력함수 fgets() 에는 버퍼가 크더라도 파일의 첫 번째 행만 읽어서 반환하지만 read()는 버퍼의 크기만큼 읽을 수 있다면 모두 읽어들인다. 그러므로 read()에서 사용할 버퍼의 크기가 파일 보다 크다면 파일의 모든 내용을 읽어 들이게 된다.	

◑ **write() 함수**

기 능	open()함수오 열기를 한 파일에 쓰기를 한다.	
헤 더	#include <unistd.h>	
선 언	ssize_t write(int fd, const void *buf, size_t n);	
형식인자	fd	파일 디스크립터
	buf	파일에 쓰기를 할 데이터를 가리키는 포인터
	n	쓰기를 할 문자수(바이트 수)
리 턴	성공시	쓰기를 한 문자의 수(바이트 수)
	실패시	-1
설 명		

▶ **파일폐쇄함수**

헤더	함수원형
stdio.h	int fclose(FILE *fp);
stdio.h	int floseall();
unistd.h	int close(int fd);

◑ **fclose() 함수**

기 능	fopen()함수에 의해 개방된 파일을 닫는다.	
헤 더	#include <stdio.h>	
선 언	int fclose(FILE *fp);	
형식인자	fp	닫을 파일 구조체 포인터
리 턴	성공시	0
	실패시	EOF
설 명	fclose()함수는 stream으로 연결되어 있는 파일의 스트림 연결을 모두 끊는다. 만약 닫고자 하는 스트림이 출력을 위해 열려져 있던 스트림이라면 fflush()함수를 이용하여 버퍼에 남아 있는 데이터를 모두 출력하여야 파일에 저장된다. 만약 열려져 있지 않은 파일이나 이미 닫혀진 파일에 대하여	

	fclose()함수를 호출하는 경우 어떤일이 일어날지는 알 수 없다. 프로그램이 세그멘테이션 오류로 종료될 수 도 있고 계속 작동될 수 도 있다. 또한 fclose()함수는 실패시 errno를 설정하게 되는데 errno의 값은 다음과 같다. EBADF : 잘못 연결된 stream

◑ **fcloseall() 함수**

기 능	열려져 있는 모든 파일 스트림을 닫는다.	
헤 더	#include <stdio.h>	
선 언	int fcloseall(void);	
리 턴	성공시	닫혀진 스트림의 총 수
	실패시	매크로 EOF(-1)
설 명	이 함수는 열려져 있는 모든 파일 스트림을 닫는다. 이때 stdin, stdout, stderr, stdaux 등 기정의 스트림은 폐쇄하지 않는다.	

◑ **close() 함수**

기 능	파일 디스크립터의 복사본을 생성한다.	
헤 더	#include <unistd.h>	
선 언	int close(int fd);	
형식인자	fd	파일 디스크립터
리 턴	성공시	0
	실패시	매크로 EOF(-1)
설 명	이 함수는 open()함수에 의해 열려진 파일을 닫는다.	

▶ **파일 포인터 함수**

헤더	함수원형
stdio.h	int fgetpos(FILE *fp, fpost_t *position);
stdio.h	int fsetpos(FILE *fp, const fpos_t *pos);
stdio.h	long ftell(FILE *fp);

stdio.h	int fseek(FILE *fp, long offset, int whence);
stdio.h	void rewind(FILE *fp);

◑ **fgetpos() 함수**

기 능	파일구조체의 위치지정자가 가리키는 위치를 구한다.	
헤 더	#include <stdio.h>	
선 언	int fgetpos(FILE *fp, fpos_t *pos);	
형식인자	fp	파일 구조체 포인터
	pos	fpos_t 객체를 가리키는 포인터
리 턴	성공시	0
	실패시	0이 아닌 값
설 명	파일구조체의 위치 지정자(position indicator)가 가리키는 위치를 pos에 저장한다. 따라서 인자로 전달되는 pos는 fpos_t 의 형을 가리키는 포인터 형태로 사용되어야 하며, 거의 대부분 fsetpos()함수의 인자로만 사용하게 된다. 만일 파일 위치 지정자의 값을 정수형 데이터로 얻고 싶다면 ftell()함수를 호출하면 된다.	

◑ **fsetpos() 함수**

기 능	파일 구조체가 가리키는 위치 지정자를 설정한다.	
헤 더	#include <stdio.h>	
선 언	int fsetpos(FILE *fp, const fpos_t *pos);	
형식인자	fp	작업을 수행할 대상 파일 구조체 포인터
	pos	fpos_t 객체를 가리키는 포인터로 반드시 이전 fgetpos()함수에 의해 얻어진 값을 지니고 있어야 한다.
리 턴	성공시	0
	실패시	0이 아닌 값
설 명	파일 구조체의 위치 지정자를 새로운 위치로 변경한다. 위 함수에서 인자로 전달되는 pos는 fpos_t 객체를 가리키는 포인터로 반드시 이전의 fgetpos()함수에 의해 얻어진 값을 가지고 있어야 한다.	

◑ **ftell() 함수**

기 능	파일 구조체의 위치지정자 값을 구한다.	
헤 더	#include <stdio.h>	
선 언	long ftell(FILE *fp);	
형식인자	fp	작업을 수행할 파일 구조체 포인터
리 턴	성공시	현재 위치 지정자가 가리키는 값
	실패시	NULL
설 명	이 파일은 파일 구조체에 설정되어 있는 위치 지정자의 값을 구하는 함수이다. 이 함수는 성공시 위치 지정자가 가리키는 값을 반환하며, 실패시에는 NULL값을 반환하고 errno를 설정한다.	

◑ **fseek() 함수**

기 능	파일 구조체의 위치지정자의 위치를 조정한다.	
헤 더	#include <stdio.h>	
선 언	int fseek(FILE *fp, long offset, int origin);	
형식인자	fp	작업을 수행할 파일 구조체 포인터
	offset	origin으로부터 얼마나 떨어진 위치에 설정 할 것인가에 대한 값
	origin	오프셋이 더해지는 기준위치 SEEK_SET : 파일의 시작 SEEK_CUR : 현재 위치 SEEK_END : 파일의 끝
리 턴	성공시	0
	실패시	0이 아닌 값
설 명	이 파일은 파일 구조체 포인터의 위치 지정자를 조정하는 함수이다. 이 함수를 호출 한 이후에는 파일 끝 지정자(End of File indicator)가 초기화 되고, 이전에 만일 ungetc()함수를 호출하였다면 이로 인한 모든 효과는 사라지게 된다. 주의할 점은 텍스트 파일에 대하여 fseek()함수를 사용할 때, offset값으로 0이 아닌 값, 혹은 ftell()함수에 의해 반환된 값을 사용할 때에는 일부 플랫폼에서는 약간의 문제가 생겨서 예상치 못한 위치에 위치지정자가 설정되는 경우가 있다.	

	fseek()함수는 스트림이 읽기 및 쓰기 형식으로 열려있을 때, 이 함수를 호출하므로써 읽기 및 쓰기모드로 전환할 수 있다.

◑ **rewind() 함수**

기 능	파일의 위치 지정자를 맨 처음으로 설정한다.	
헤 더	#include <stdio.h>	
선 언	void rewind(FILE *fp);	
형식인자	fp	파일 구조체 포인터
설 명	파일 구조체의 위치 지정자를 맨 처음으로 설정한다. 이 함수는 아래와 같은 역할을 수행한다. fseek(fp, 0L, SEEK_SET); 단 fseek()함수와는 달리 rewind()함수는 오류지정자를 초기화 한다. 따라서 파일이 읽기 및 쓰기 형식으로 열려있을 경우, rewind()함수를 호출함으로써 읽기에서 쓰기모드로 쓰기에서 읽기모드로 변경할 수 있다.	