

## 4. 함수 (Function)



교수 김 영 탁

영남대학교 정보통신공학과

(Tel : +82-53-810-2497; Fax : +82-53-810-4742

<http://antl.yu.ac.kr/>; E-mail : ytkim@yu.ac.kr)

# Outline

- ◆ 모듈화 프로그래밍과 함수
- ◆ 함수 정의, 호출과 반환
- ◆ 함수 원형 (Function Prototype)
- ◆ 라이브러리 함수 (Library functions)
- ◆ 지역변수 (local variable), 전역변수 (global variable)
- ◆ 정적 지역 변수 (static local variable)
- ◆ 범위 (scope), 생존기간
- ◆ 연결 (link), extern 선언
- ◆ 재귀함수 (recursive function)
- ◆ 동적프로그래밍 (dynamic programming)



# 모듈화 프로그래밍과 함수

# 모듈화 프로그래밍의 개념

## ◆ 모듈(module)

- 독립되어 있는 프로그램의 일부분

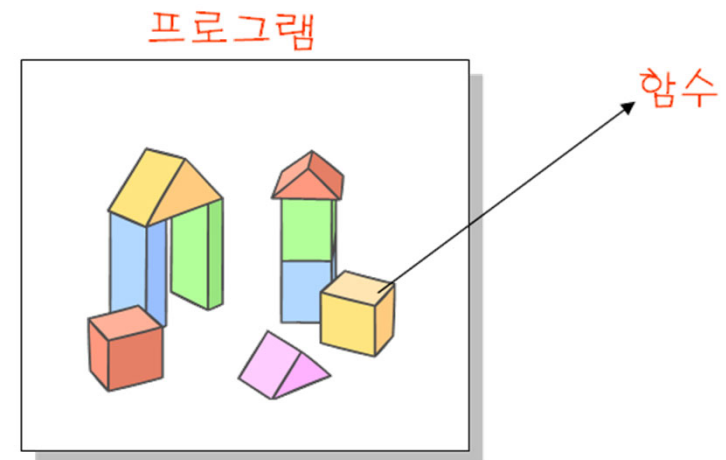
## ◆ 모듈화 프로그래밍

- 모듈 개념을 사용하는 프로그래밍 기법
- 프로그램에 포함되는 기능들을 모듈 별로 나누어 설계 및 구현
- 일부 모듈은 기존에 이미 개발되어 있는 모듈이나 라이브러리를 사용

## ◆ 모듈화 프로그래밍의 장점

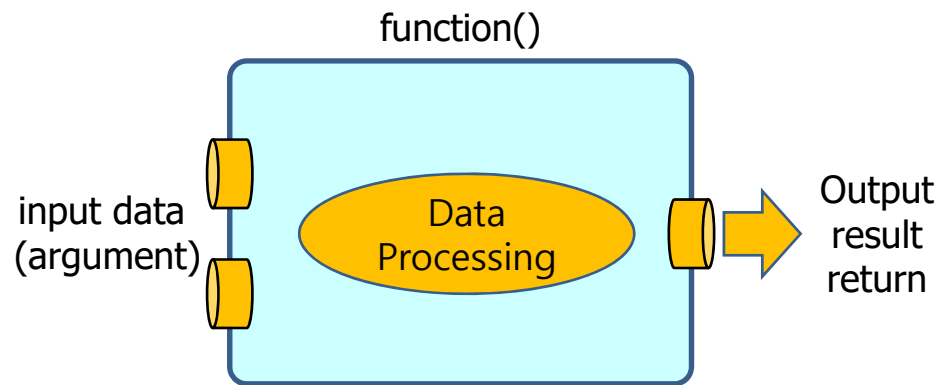
- 각 모듈들은 독자적으로 개발 가능
- 다른 모듈과 독립적으로 변경 가능
- 유지 보수가 쉬워진다.
- 모듈의 재사용 가능

## ◆ C에서는 모듈 == 함수



# 함수의 개념

- ◆ 함수(function) : 특정한 작업을 수행하는 독립적인 모듈
- ◆ 함수 호출(function call) : 함수를 호출하여 사용하는 것
- ◆ 함수는 입력을 받으며 출력을 생성한다.



함수는 특정한 작업을 수행하는 독립적인 모듈이며, 데이터(인수)를 전달 받아 처리하고, 그 결과를 반환합니다.



# 함수의 필요성

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

```
int main(void)
```

```
{
```

```
    for(int i = 0; i < 100; i++)  
        printf("*");  
    printf("\n");
```

```
    ...
```

```
    for(int i = 0; i < 100; i++)  
        printf("*");  
    printf("\n");
```

```
    ...
```

```
    for(int i = 0; i < 100; i++)  
        printf("*");  
    printf("\n");
```

```
    return 0;
```

```
}
```

한줄에 100개의 \*을 출력하는 코드

한줄에 100개의 \*을 출력하는 코드

한줄에 100개의 \*을 출력하는 코드



# 함수의 필요성

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

```
void print_star_line()
```

```
{
```

```
    for(int i = 0; i < 100; i++)
```

```
        printf("*");
```

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

```
}
```

```
int main(void)
```

```
{
```

```
    print_star_line();
```

```
    ...
```

```
    print_star_line();
```

```
    ...
```

```
    print_star_line();
```

```
    return 0;
```

```
}
```

함수를 한번 정의 한 후  
여러 번 호출하여  
실행이 가능하다.



# 함수 사용의 장점과 단점

## ◆ 함수 사용의 장점

- 함수를 사용하면 코드가 중복되는 것을 막을 수 있다.
- 한번 작성된 함수 (모듈)는 여러 번 재사용할 수 있다.
- 함수를 사용하면 전체 프로그램을 모듈로 나눌 수 있어서 개발 과정이 쉬워지고 보다 체계적이 되면서 유지보수도 쉬워진다.

## ◆ 함수 사용의 단점

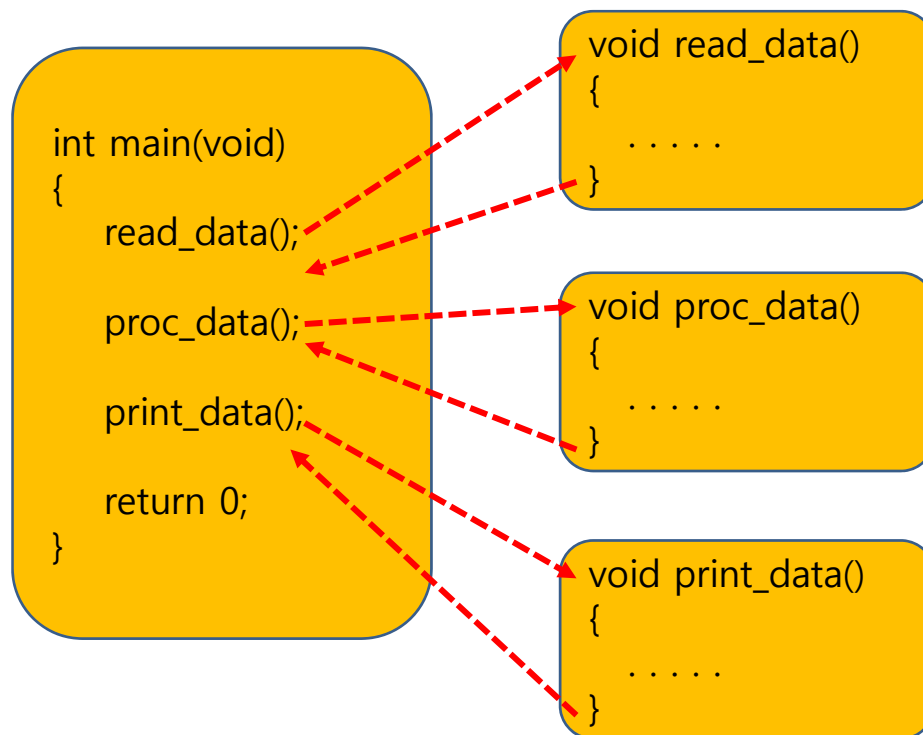
- 함수 호출을 할 때 마다 운영체제가 함수에서 사용되는 지역 변수들의 준비와 인수 (argument) 전달 등을 처리해야 하므로 부담이 발생한다.
- 따라서 너무 작은 단위의 함수를 구성하여 자주 호출하는 경우 성능에 문제가 발생할 수 도 있다.





# 함수들의 연결

- ◆ 프로그램은 여러 개의 함수들로 이루어진다.
- ◆ 함수 호출을 통하여 서로 서로 연결된다.
- ◆ 제일 먼저 호출되는 함수는 **main()**이다.



main() 함수에서는 전체 기능을 구현하지 않고, 필요한 기능을 해당 함수들을 호출하여 차례로 수행합니다.



# 함수의 종류

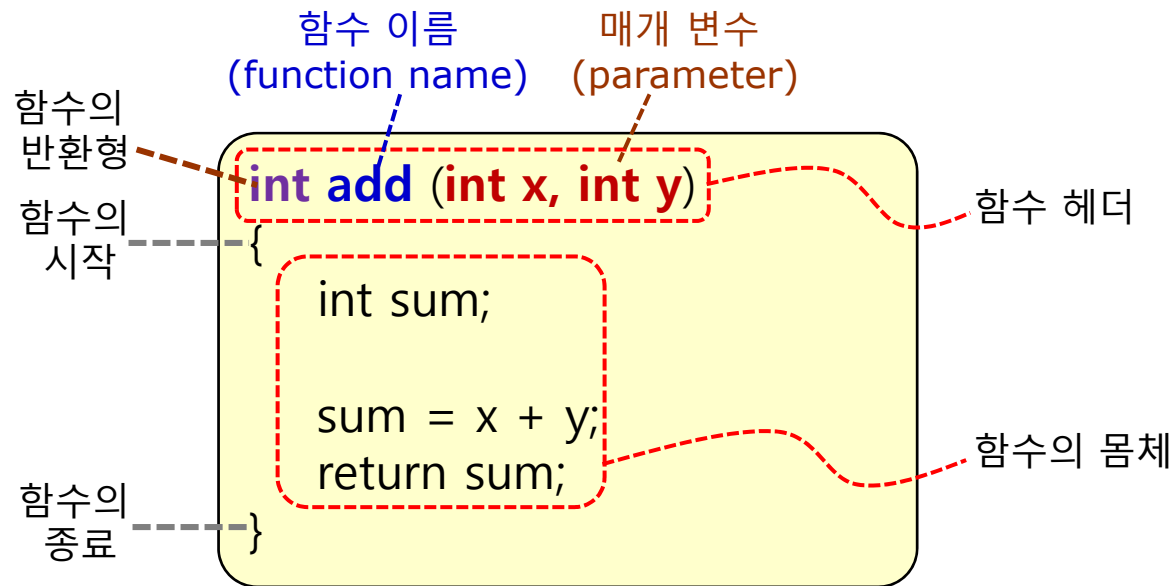
## ◆ 사용자 정의함수 vs. Library 함수

함수 (function)	개발 및 제공	예
라이브러리 함수 (library function)	<ul style="list-style-type: none"><li>프로그래밍언어에서 기본적인 함수로 제공</li></ul>	<ul style="list-style-type: none"><li>scanf(), printf()</li><li>time()</li><li>rand()</li></ul>
사용자 정의 함수 (user defined function)	<ul style="list-style-type: none"><li>사용자가 직접 설계 및 구현</li><li>필요에 따라 필요한 함수를 구현</li></ul>	<ul style="list-style-type: none"><li>mtrxAdd()</li><li>mtrxSubtract()</li><li>mtrxMultiply()</li></ul>



# 함수의 구조

## ◆ 함수의 구조



# 반환 데이터 유형 (return data type)

반환형

int

double

void

square()

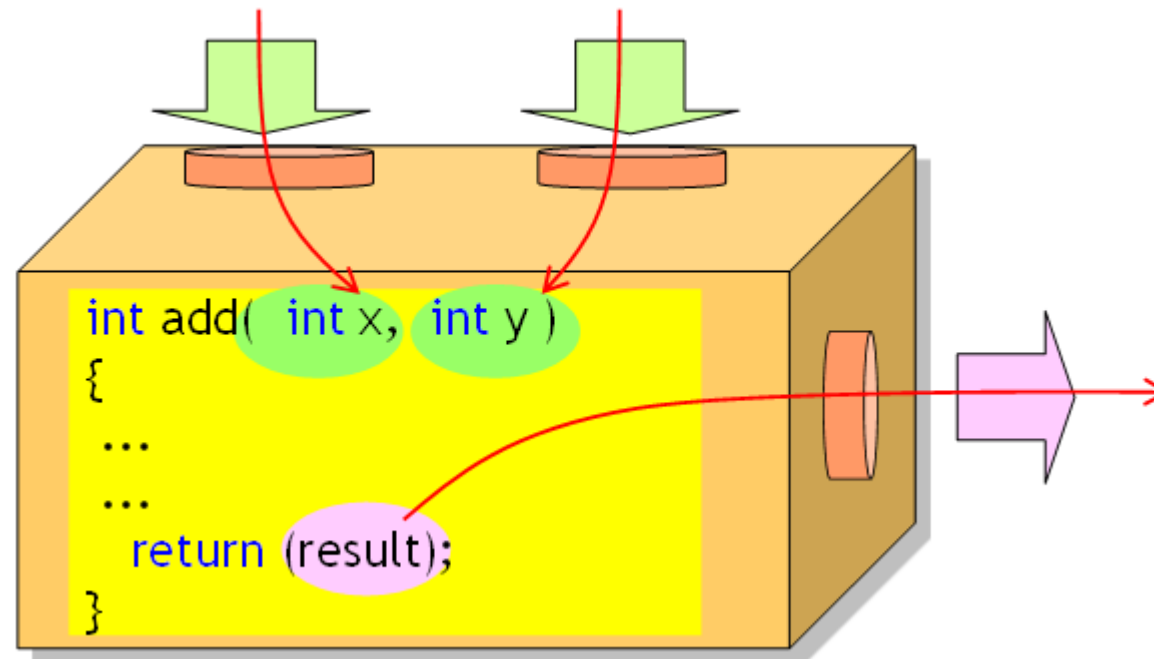
compute\_average()

set\_cursor\_type()

// int 형의 값을 반환한다.

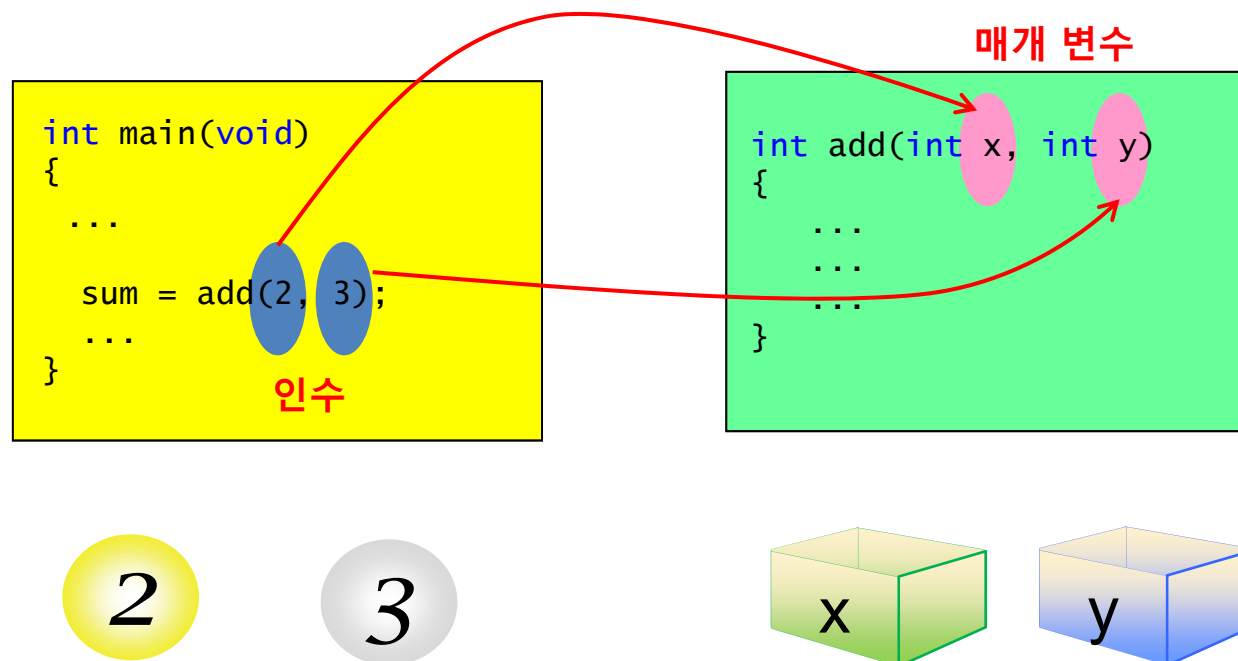
// double 형의 값을 반환한다.

// 반환값이 없는 함수



# 매개 변수 (parameter), 인수 (argument)

- ◆ **매개 변수(parameter):** 형식 인수, 형식 매개 변수라고도 한다.
  - 함수 원형 (function prototype) 선언에서 형식 인수 (형식 매개변수) 데이터 유형 지정
- ◆ **인수(argument):** 실인수, 실매개 변수라고도 한다.
  - 프로그램 실행단계에서 실제 함수 호출에 전달되는 데이터

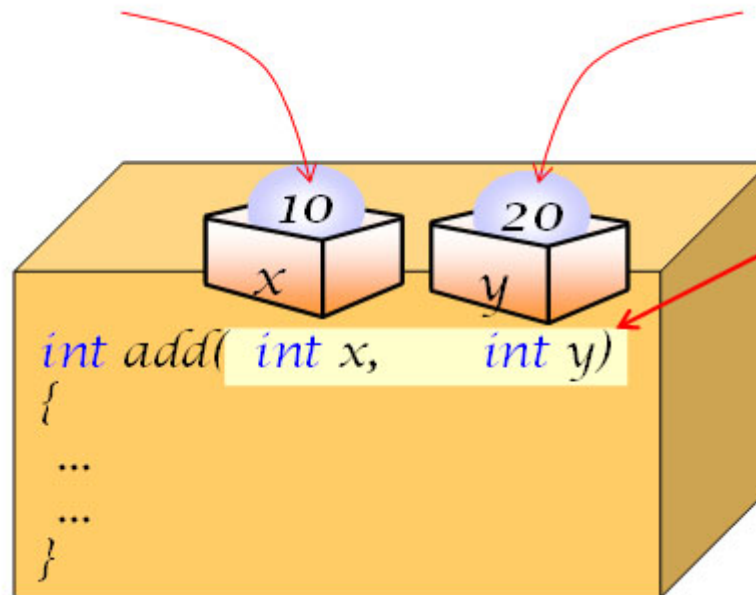


# 인수 (argument), 매개 변수 (parameter)

```
int square(int n)
double compute_average(double x, double y)
void get_cursor_type(void)
```

// 정수를 제공하는 함수  
// 평균을 구하는 함수  
// 커서의 타입을 반환하는 함수

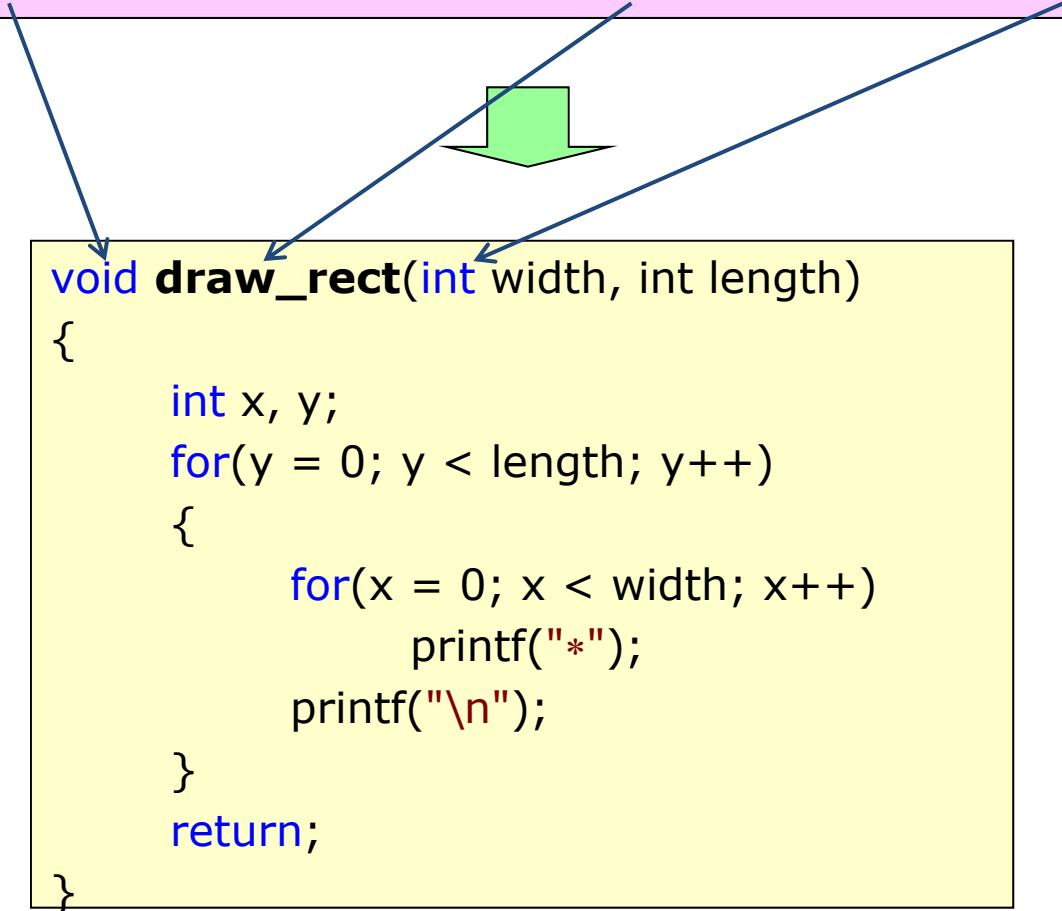
매개 변수



# 예제

## ◆ 별표 기호를 이용하여 직사각형을 그리는 함수

return type: void / function name: draw\_rect / parameters: int width, int length



```
void draw_rect(int width, int length)
{
    int x, y;
    for(y = 0; y < length; y++)
    {
        for(x = 0; x < width; x++)
            printf("*");
        printf("\n");
    }
    return;
}
```



# 기본 인수값 (default argument value) 지정

- ◆ 함수호출에서 인수의 값이 지정되지 않았을 때, 사전에 기본 인수 값으로 설정된 값을 사용
- ◆ 함수의 선언 및 함수 원형(prototype)에서 설정
  - **void calcVolume(int length, int width = 1, int height = 1);**
    - Last 2 arguments are defaulted
  - Possible calls:
    - **calcVolume(2, 4, 6);** //All arguments supplied
    - **calcVolume(3, 5);** //height defaulted to 1
    - **calcVolume(7);** //width & height defaulted to 1

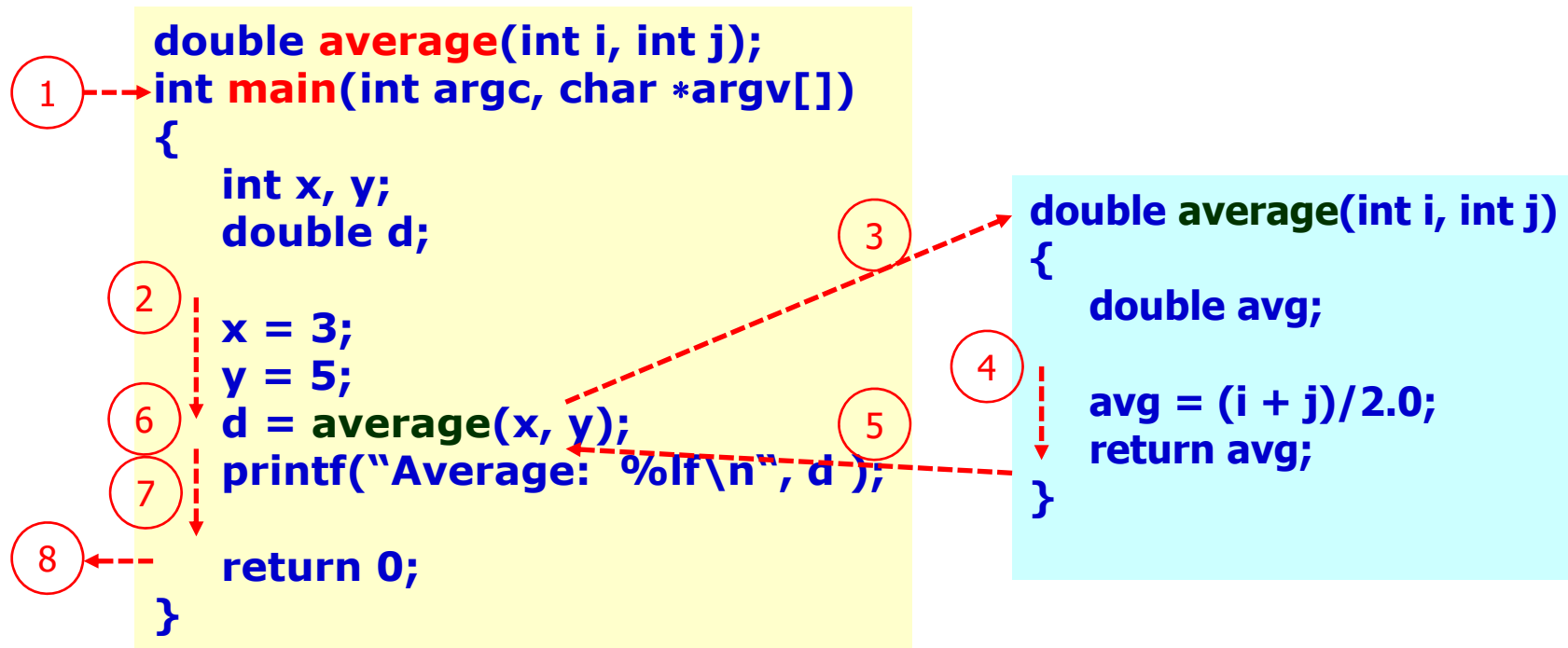




# 함수 호출과 반환

## ◆ 함수 호출(function call):

- 함수를 사용하기 위하여 함수의 이름을 적어주는 것
- 함수안의 문장들이 순차적으로 실행된다.
- 문장의 실행이 끝나면 호출한 위치로 되돌아 간다.
- 결과값을 전달할 수 있다.



# 함수 호출 시 Call-by-Value에 의한 인수 전달

```
double average(int i, int j);  
void main(int argc, char *argv[])  
{
```

```
    int x, y;  
    double d;
```

```
    x = 3;  
    y = 5;  
    d = average(x, y);  
    printf("Average: %lf\n", d );
```

```
}
```

```
double average(int i, int j)  
{
```

```
    double avg;
```

```
    avg = (i + j)/2.0;  
    return avg;
```

```
}
```

Function  
Call:  
main()

stack frame for main( )  
- arguments: int argc, char \*argv[]  
- local variables: int x, int y, double d

Memory Stack

Function  
Call:  
average()

stack frame for average( )  
- arguments: int i, int j  
- local variables: double avg

copy, copy

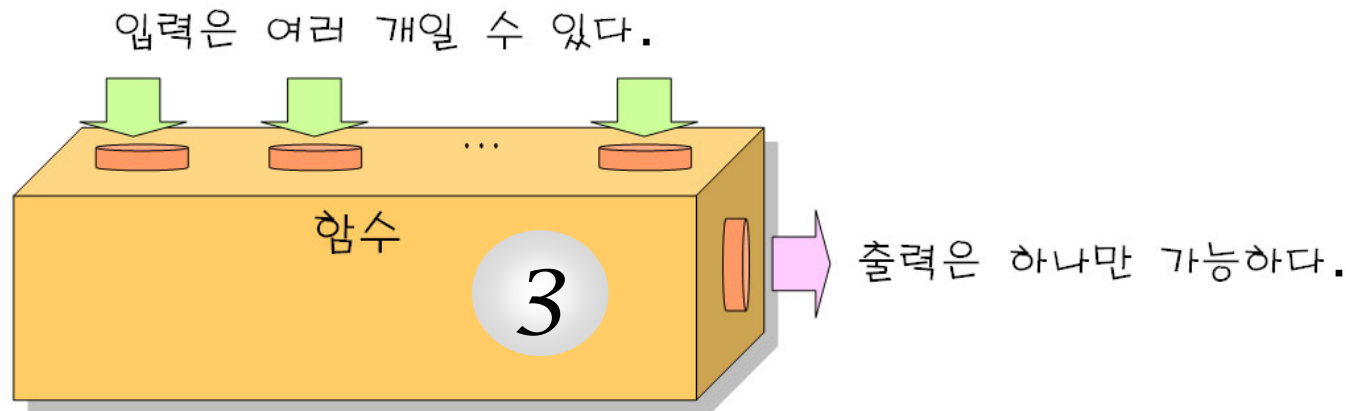
stack frame for main( )  
- arguments: int argc, char \*argv[]  
- local variables: int x, int y, double d

Memory Stack



# 함수 실행 결과의 반환 (return)

- ◆ 반환 값(return value): 호출된 함수가 호출한 곳으로 작업의 결과값을 전달하는 것
- ◆ 인수는 여러 개가 가능하나 반환 값은 하나만 가능



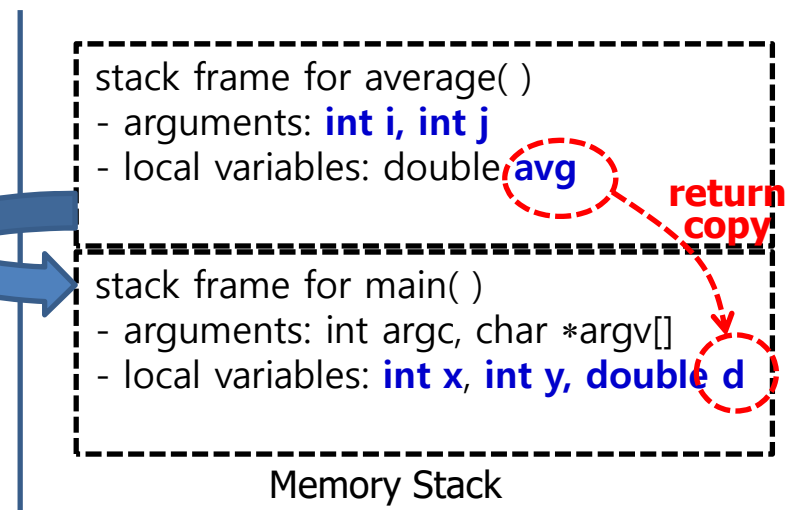
```
return 0;  
return(0);  
return x;  
return x*x+2*x+1;
```

# 함수 실행 결과의 Return-by-Value에 의한 전달

```
double average(int i, int j);
void main(int argc, char *argv[])
{
    int x, y;
    double d;

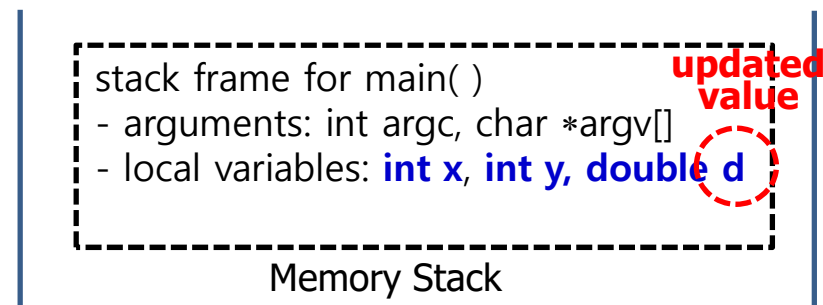
    x = 3;
    y = 5;
    d = average(x, y);
    printf("Average: %lf\n", d );
}
```

Function  
Return



```
double average(int i, int j)
{
    double avg;

    avg = (i + j)/2.0;
    return avg;
}
```



# 함수 원형 (Function Prototype)

## ◆ 함수 호출에 대한 컴파일에서 필요한 정보

- 그 함수에 대한 정보 (함수 이름, 전달되는 인수 목록, 반환 자료형 등)

## ◆ 함수 원형(function prototype)

- 미리 컴파일러에게 함수에 대한 정보를 알려주는 역할 수행

반환형   함수이름(매개변수1, 매개변수2, ... );

(예)

- int get\_integer(void);
- int combination(int n, int r);

(예)

- int get\_integer(void);
- int combination(int, int);

매개변수의 자료형만  
적어주어도 됨!



# 함수 원형의 사용 예

- ◆ 함수 원형(function prototype): 컴파일러에게 함수에 대하여 미리 알리는 것

```
int compute_sum(int n);  
int main(void)  
{  
    int sum;  
    sum = compute_sum(100);  
    printf("sum=%d \n", sum);  
}  
int compute_sum(int n)  
{  
    int i;  
    int result = 0;  
    for(i = 1; i <= n; i++)  
        result += i;  
    return result;  
}
```

compute\_sum()을  
호출 하기 위하여 어떤  
인수를 전달하고,  
결과값을 어떻게  
받아오지 ?



컴파일러



# 함수 원형과 헤더 파일

## ◆ 일반적으로 헤더 파일에 함수 원형이 선언되어 있음

```
/* 두개의 숫자의 합을 계산하는 프로그램 */
#include <stdio.h>

int main(void)
{
    int n1;    /* 첫번째 숫자 */
    int n2;    /* 두번째 숫자 */
    int sum;   /* 두개의 숫자의 합을 저장 */

    printf("첫번째 숫자를 입력하시오:");
    scanf("%d", &n1);

    printf("두번째 숫자를 입력하시오:");
    scanf("%d", &n2);

    sum = n1 + n2;
    printf("두수의 합: %d", sum);

    return 0;
}
```

```
/**
 *stdio.h - definitions/declarations for
 *standard I/O routines
 *
 ****/

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

stdio.h



## **라이브러리 함수 (Library Functions) (1)**



# 라이브러리 함수

## ◆ 라이브러리 함수(library function): 컴파일러에서 제공하는 함수

라이브러리 분류	라이브러리 함수 예
표준 입출력	scanf(), printf(), getchar(), putchar(), gets(), puts()
시간 관련	time(), localtime()
난수 생성	rand(), srand()
수학 연산	sin(), cos(), sqrt(), pow()
문자열 (string) 관련	strlen(), strcat(), strcpy(), strcmp()
파일 입출력 관련	fopen(), fclose(), fscanf(), fprintf(), fget()
시스템 I/O 관련	Beep()
비정상 상황 오류 처리	exception
알고리즘 관련	sort(), search()
스레드 관련	CreateThread(), _beginthreadex(), _endthreadex(), join() WaitForSingleObject(), TerminateThread(), CloseHandle(),
임계구역 (critical section)	InitializeCriticalSection(), EnterCriticalSection(), LeaveCriticalSection(), DeleteCriticalSection(), mutex()
인터넷 통신 관련	socket()



# 표준입출력 라이브러리 함수

## ◆ 표준입출력

표준입출력 함수 함수원형	기능
<code>printf("format-string", 변수1, 변수2, ...)</code>	지정된 포맷으로 표준 출력장치 (모니터)로 출력
<code>scanf("format-string", &amp;변수1, &amp;변수2, ...)</code>	지정된 포맷으로 표준 입력장치 (키보드)로 부터 입력
<code>int getchar()</code>	문자 한자 단위로 입력
<code>int putchar(char 문자)</code>	문자 한자 단위로 출력
<code>char *gets(char *str)</code>	(" \n"로 끝나는) 문자열 단위 입력
<code>int puts(char *str)</code>	(" \n"로 끝나는) 문자열 단위 출력



# scanf() 함수의 포맷 지정

포맷 문자 (Format Character)	입력 데이터 유형 (input data type)	입력 포맷
%d	int	signed decimal integer
%i	int	signed decimal integer
%u	unsigned int	unsigned decimal integer
%o	unsigned int	unsigned octal integer
%x	unsigned int	unsigned hexadecimal integer
%c	char	character
%s	char *	string indicated by a character pointer
%p	void *	address value of the pointer
%f	float	signed floating point number
%lf	double	signed double precision floating point number
%e, %g	float, double	signed floating point number



# printf() 포맷 지정

포맷 문자 (Format Character)	출력 데이터 유형 (Output data type)	출력 포맷 (Output)
%d	int	signed decimal integer
%u	unsigned int	unsigned decimal integer
%o	unsigned int	unsigned octal integer
%x, %X	unsigned int	unsigned hexadecimal integer
%f	float	floating point numbers in decimal format
%lf	double	double precision floating point numbers in decimal format
%e, %E	float, double	floating point numbers in scientific format (e.g., 1.2345e-001 or 1.0E-20)
%g, %G	float, double	selects %f or %e according to the value
%c	char	character
%s	char *	string indicated by a character pointer
%p	void *	address value of the pointer
%n	int *	address value of the pointer



# printf()에서의 출력공간 및 정렬 지정

포맷 문자 (Format Character)	출력 데이터 유형 (Output data type)	출력 포맷 (Output)
%8d	int	10진수를 8칸에 오른쪽 맞춤으로 출력
%10.2f, %10.2lf	float, double	실수 (float, double)을 10칸에 소수점 이하 2자리까지 출력
%8o	unsigned int	8진수를 8칸에 오른쪽 맞춤으로 출력
%8x, %8X	unsigned int	16진수를 8칸에 오른쪽 맞춤으로 출력
%#d	int	10진수를 출력 (10진수의 경우 별도의 prefix없음)
%#o	unsigned int	8진수를 prefix (o)과 함께 출력
%#x, %#X	unsigned int	16진수를 prefix (0x 또는 0X)과 함께 출력
%#08d	int	10진수를 8칸에 오른쪽 맞춤으로 출력하며, 앞의 빈자리에는 0을 채워줌 (10진수의 경우 별도의 prefix없음)
%#09o	unsigned int	8진수를 prefix (o)과 함께 9칸에 오른쪽 맞춤으로 출력하며, 앞의 빈자리에는 0을 채워줌
%#010x, %#010X	unsigned int	16진수를 prefix (0x 또는 0X)과 함께 10칸에 오른쪽 맞춤으로 출력하며, 앞의 빈자리에는 0을 채워줌
%-8d	int	10진수를 8칸에 왼쪽 맞춤으로 출력
%+8d	int	10진수를 8칸에 오른쪽 맞춤으로 + 부호와 함께 출력
%20s	char *	문자열을 20칸에 오른쪽 맞춤으로 출력
%-20s	char *	문자열을 20칸에 왼쪽 맞춤으로 출력



# printInt\_inBits()

```
#define NUM_BITS_INT 32
#define BIT_MASK 0x01

void printInt_Bits(int d)
{
    unsigned long bit;

    for (int n = (NUM_BITS_INT - 1); n >= 0; n--)
    {
        bit = (d >> n) & BIT_MASK;
        printf("%d", bit);
        if ((n % 8) == 0)
            printf(" ");
    }
}
```



# 시간 관련 함수

분류	함수 원형과 인자	함수 설명
시간 계산	<code>time_t time(time_t *timeptr);</code>	1970년 1월 1일 자정부터 경과된 현재 시간을 초단위로 계산
시간을 문자열로 변환	<code>char *asctime(struct tm *time);</code>	구조체 tm형식의 시간을 문자열로 변환
	<code>char *ctime(time_t *time);</code>	함수 time()로부터 계산된 현재 시간을 문자열로 변환
시간을 구조체로 변환	<code>struct tm *localtime(time_t *time);</code>	지역 시간(local time)을 구조체 tm의 형식으로 가져오는 함수
	<code>struct tm *gmtime(time_t *time);</code>	Greenwich Mean Time(GMT)을 구조체 tm 형식으로 가져옴
시간 차이 계산	<code>clock_t clock(void);</code>	clock tick으로 경과된 시간
	<code>double difftime(time_t time2, time_t time1);</code>	두 시간의 차이를 초단위로 계산
시간 지연	<code>void Sleep(unsigned millisecond);</code> <code>void delay(unsigned millisecond);</code>	인자가 지정하는 만큼의 밀리초 (1/1000초) 단위의 시간을 지연



# 시간 관련 함수의 사용 예

```
/* main() for Date_and_Time.c */

#include <stdio.h>
#include <time.h>

void main()
{
    time_t currentTime; // time_t 구조체 변수
    struct tm *info;    // tm 구조체 포인터

    time(& currentTime);
    info = localtime(& currentTime);
    printf("Current local time and date: %s", asctime(info));
}
```

```
Current local time and date: Mon Mar 15 10:56:55 2021
```





# micro-second 단위의 실행 시간 측정을 위한 Windows 라이브러리 함수 - QueryPerformanceCounter()

## ◆ micro-second 단위의 경과 시간 측정

- Windows 운영체제에서 제공하는 Performance Counter를 사용
- Performance Counter는 CPU의 clock tick 단위로 경과시간 측정 가능
- CPU는 2GHz 이상의 고속 clock frequency를 사용하므로 1 micro-second ( $10^{-6}$  second)미만의 정밀한 경과시간 측정 가능
- 참고: <https://www.pluralsight.com/blog/software-development/how-to-measure-execution-time-intervals-in-c-->

## ◆ Performance Counter 관련 라이브러리 함수

- LARGE\_INTEGER freq, t : LARGE\_INTEGER는 Windows 운영체제에서 사용하는 64비트 정수
- QueryPerformanceFrequency(&freq): performance counter 주파수 (frequency)를 기록 (단위: ticks\_per\_second)
- QueryPerformanceCounter(&t): performance counter 값을 기록



# QueryPerformanceCounter() 예제

```
#include <Windows.h>
```

```
....  
int function_to_be_tested(int array[], int size);  
// QueryPerformanceFrequency(LARGER_INTEGER *freq);  
// QueryPerformanceCounter(LARGER_INTEGER *time);
```

```
int main()  
{
```

```
    LARGE_INTEGER freq, t_1, t_2  
    LONGLONG t_diff;  
    double elapsed_time_us;  
    int *array, array_size;
```

```
    array = (int *)calloc(array_size, sizeof(int)); // 동적 배열 생성
```

```
    ....
```

```
    QueryPerformanceFrequency(&freq);  
    QueryPerformanceCounter(&t_1);
```

```
    function_to_be_tested(array, array_size); // 경과시간 측정 대상 함수의 실행
```

```
    QueryPerformanceCounter(&t_2);  
    t_diff = t_2.QuadPart - t_1.QuadPart;  
    elapsed_time_us = ((double) t_diff / freq.QuadPart)*1000000; // in micro-second
```

```
    printf("It took %f [micro-seconds] to perform the function with %d integer data array.",  
           elapsed_time_us, array_size);
```

```
    ....
```

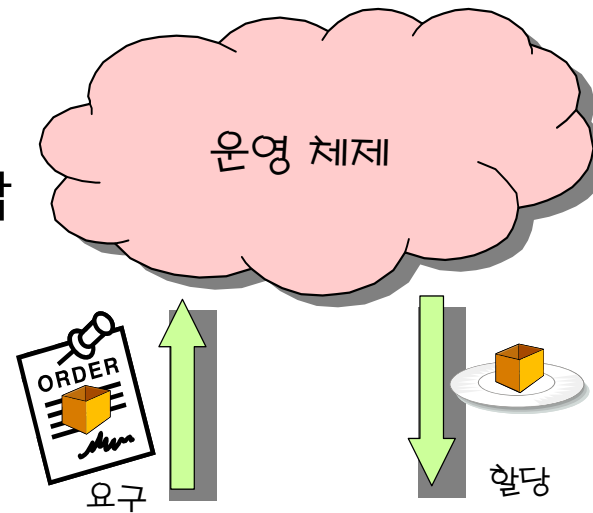
```
}
```



# 동적 (dynamic) 메모리 할당, 동적 배열 생성

## ◆ 동적 메모리 할당

- 실행 도중에 동적으로 메모리를 할당 받는 것
- 사용이 끝나면 시스템에 메모리를 반납
- `//int score[100];` //정적 배열 대신,  
`int *score;`  
`score = (int *) calloc(100, sizeof(int));`  
로 동적으로 메모리를 할당하여,  
배열로 사용 할 수 있음
- 필요한 만큼만 할당을 받고 메모리를  
매우 효율적으로 사용
- `malloc()`, `calloc()` 계열의 라이브러리  
함수를 사용



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

int main(void)
{
    int *p;
    p = (int *)malloc( sizeof(int) );
    ...
}
```

프로그램



# 동적 메모리 블록 할당 및 반환 관련 <stdlib.h> 라이브러리 함수

분류	함수 원형과 인수	기능
동적 메모리 블록 할당 및 반환 <stdlib.h>	<code>void* malloc(size_t size)</code>	지정된 size 크기의 메모리 블록을 할당하고, 그 시작 주소를 void pointer로 반환
	<code>void *calloc(size_t n, size_t size)</code>	size 크기의 항목을 n개 할당하고, 0으로 초기화 한 후, 그 시작 주소를 void pointer로 반환
	<code>void *realloc(void *p, size_t size)</code>	이전에 할당받아 사용하고 있는 메모리 블록의 크기를 변경 p는 현재 사용하고 있는 메모리 블록의 주소, size는 변경하고자 하는 크기; 기존의 데이터 값은 유지된다
	<code>void free(void *p)</code>	동적 메모리 블록을 시스템에 반환; p는 현재 사용하였던 메모리 블록 주소



# 동적 배열 생성

## ◆ 배열을 위한 메모리 할당 방법

- 지역 변수로 자동 할당

```
int score[100];  
score[10] = 123;
```

- 지역 변수로 배열을 생성하는 경우, 크기에 제한이 있음

## ◆ 동적 배열

- 동적 메모리 할당

```
int *score, array_size;  
printf("input array_size : ");  
scanf("%d", &array_size);  
score = (int *) calloc(array_size , sizeof(int));  
score[10] = 123;
```

- 동적 메모리 할당으로 배열을 생성하는 경우,  
더 큰 배열을 사용할 수 있음



# 동적 배열 생성 예제

```
#include <stdio.h>
#include <stdlib.h> // 동적메모리 할당 calloc()

int main(void)
{
    int *dyn_array;
    int array_size, i;

    printf("Input array_size : ");
    scanf("%d", &array_size);
    dyn_array = (int *)calloc( array_size, sizeof(int) );
    if( dyn_array == NULL ) // 반환값이 NULL인지 검사
    {
        printf("동적 배열 생성 오류\n");
        exit(1);
    }
    for(i=0 ; i<100 ; i++)
        dyn_array[i] = 0;
    free(dyn_array);
    return 0;
}
```

동적 배열 생성  
(동적 메모리 블록 할당)

동적 배열의 메모리 반환



## **라이브러리 함수 (Library Functions) (2)**

# 난수 (random number) 관련 라이브러리 함수

분류	함수 원형과 인자	함수 설명
난수 생성	<code>int rand()</code>	returns a pseudo-random number in the range of 0 to RAND_MAX.
	<code>#define RAND_MAX 0x7FFF</code>	RAND_MAX로 0x7FFF (32,767) 정의
	<code>void srand(unsigned int seed)</code>	Configure the seed the random number generator used by the pseudo-random number generator algorithm of the rand() function.





# 난수 (random number) 생성

## ◆ 난수(random number)

- 규칙성이 없이 임의로 생성되는 수이다.
- 난수는 암호학이나 시뮬레이션, 게임 등에서 필수적이다.

## ◆ rand()

- 난수를 생성하는 함수
- 0부터 RAND\_MAX까지의 난수를 생성  
`d = rand();`
- 실행할 때 마다 동일한 순서로 난수 발생
- base offset과 range 사용 (예: range 10, base 1)  
`d = rand() % range + base;`

## ◆ srand( seed )

- 매번 난수 생성 순서를 다르게 만들기 위하여 다른 seed를 설정



# srand(time(NULL))

◆ 매번 난수를 다르게 생성하려면 시드(seed)를 다르게 하여야 한다.

● srand( (unsigned)time(0) );

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

```
#define RANGE 45
```

```
int main( void )
```

```
{
```

```
    int i;
```

```
    srand( (unsigned)time(0) );
```

```
    for( i = 0; i < 6; i++ )
```

```
        printf("%d ", 1+rand()%RANGE );
```

```
    return 0;
```

```
}
```

seed를 설정하는 가장 일반적인 방법은 현재의 시각을 seed로 사용하는 것이다. 현재 시각은 실행할 때마다 달라지기 때문이다.



# 32,767보다 더 큰 난수의 생성

## ◆ rand() 함수의 한계

- rand() randomly generates 0 ~ RAND\_MAX (32,767) integer value
- if big random numbers (e.g., 0 ~ 500,000) are necessary, rand() cannot be used

## ◆ 32,767보다 더 큰 난수로 구성된 배열 생성

- **genBigRandArray(int mA[], int bigRandMax)**
- generates *non-duplicated* big random numbers in the range of 0 ~ bigRandMax-1, where bigRandMax can be bigger than RAND\_MAX (32,767)
- as result, the non-duplicated random numbers are contained in mA[]

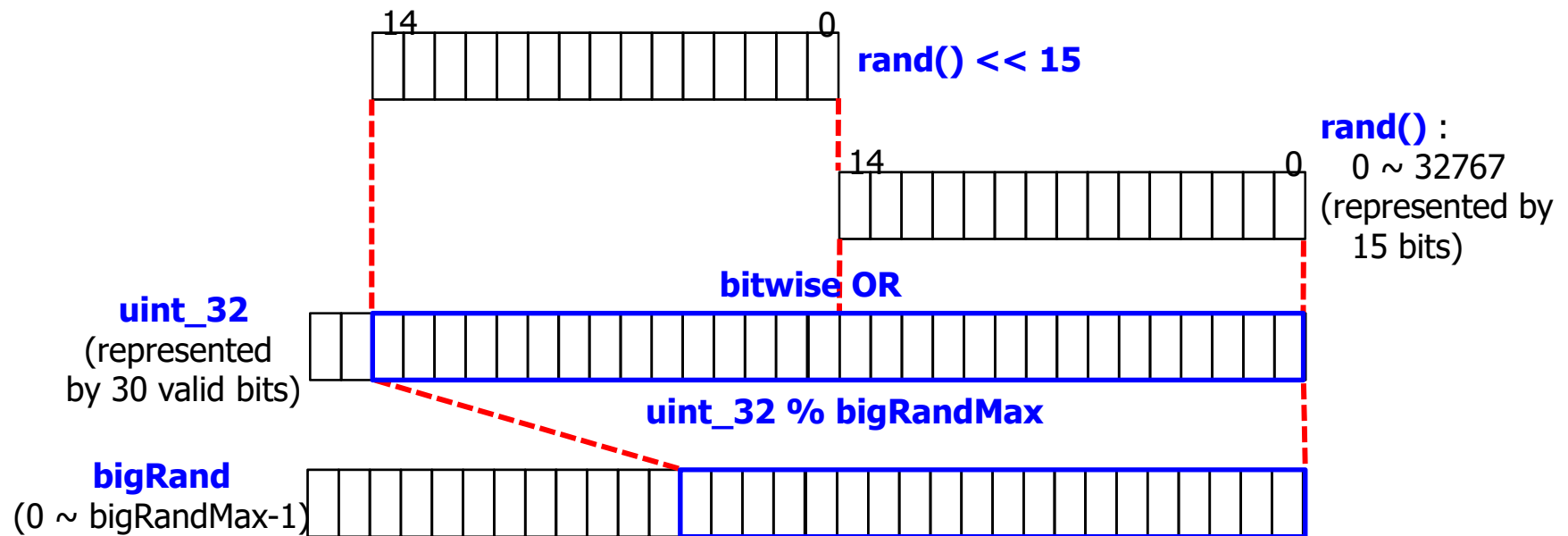


# BigRand()

## ◆ Generation of random numbers with $\text{bigRandMax} > 32767$

```
unsigned int uint_32, bigRand;
```

```
uint_32 = ((unsigned int)rand() << 15) | rand(); // bitwise left shift, bitwise OR  
bigRand = uint_32 % bigRandMax;
```



## ◆ genBigRandArray()

Procedure genBigRandArray(int mA[], int bigRandMax)

1. char \*flag;
2. unsigned int uint\_32;
3. unsigned int bigRand;
4. int count = 0;
5. srand (time(0)); // use current time as seed; needs #include <ctime>
6. flag = (char \*)malloc(sizeof(char) \* bigRandMax); // 동적배열 생성
7. while (count < bigRandMax) {
8.     uint\_32 = ((long) rand() << 15) | rand();  
       // generation by bit-wise shift of 15-bit rand short integer  
       // and bit-wise or
9.     bigRand = uint\_32 % bigRandMax;
10.    if (flag[bigRand] == 1) { // if this bigRand was already generated
11.      continue;
12.    } else {
13.      flag[bigRand] = 1; // else, use this bigRand, and mark the flag
14.      mA[count++] = bigRand;
15.    }
16. } // end while
17. END Procedure



# 유틸리티 함수 (Utility Function)

함수	설명
exit(int status)	exit()를 호출하면 호출 프로세스를 종료시킨다.
int system(const char *command)	system()은 문자열 인수를 운영 체제의 명령어 셸에게 전달하여서 실행시키는 함수이다.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h> // for _getch()
int main( void )
{
    system("dir");
    printf("Hit any key to continue :");
    _getch();
    system("cls");
    return 0;
}
```

볼륨 일련 번호: 7044-9C0C

C:\MyC\_Progs\2021 (C-Prog Book, Visual Studio 2019)\Ch 4 Function\Fig 4.50 utility function\Fig 4.50 utility function 디렉터리

```
2021-03-15 오후 03:18 <DIR> .
2021-03-15 오후 03:18 <DIR> ..
2021-03-15 오후 03:19 <DIR> Debug
2021-03-15 오후 03:19 7,223 Fig 4.50 utility function.vcxproj
2021-03-15 오후 03:18 1,000 Fig 4.50 utility function.vcxproj.filters
2021-03-15 오후 03:14 168 Fig 4.50 utility function.vcxproj.user
2021-03-15 오후 03:18 214 test_utility_functions.cpp
                4개 파일            8,605 바이트
                3개 디렉터리 719,255,396,352 바이트 남음
```

Hit any key to continue :



# 수학 라이브러리 함수 (Math Library Function)

분류	함수 원형과 인자	함수 설명
삼각 함수	double sin(double rad)	sine 값 계산, rad는 radian 단위
	double cos(double rad)	cosine 값 계산, rad는 radian 단위
	double tan(double rad)	tangent 값 계산, rad는 radian 단위
역삼각 함수	double asin(double rad)	arcsine 값 계산, rad는 radian 단위
	double acos(double rad)	arccosine 값 계산, rad는 radian 단위
	double atan(double rad)	arctangent 값 계산, rad는 radian 단위
쌍곡선 함수	double sinh(double rad)	hyperbolic sine 값 계산, rad는 radian 단위
	double cosh(double rad)	hyperbolic cosine 값 계산, rad는 radian 단위
	double tanh(double rad)	hyperbolic tangent 값 계산, rad는 radian 단위
지수 함수	double exp(double x)	$e^x$
	double log(double x)	$\log_e(x)$
	double log10(double x)	$\log_{10}(x)$
기타 수학 연산 함수	int abs(int x)	$ x $ , x는 정수 (int)
	double fabs(double x)	$ x $ , x는 실수 (double)
	double ceil(double x)	$\lceil x \rceil$ , x와 같거나 큰 정수 중, 가장 작은 정수
	double floor(double x)	$\lfloor x \rfloor$ , x와 같거나 작은 정수 중 가장 큰 정수
	double pow(double x, double y)	$x^y$ , 지수승
	double sqrt(double x)	$\sqrt{x}$ , 제곱근



# 예제

// 삼각 함수 라이브러리

#include <math.h>

#include <stdio.h>

여러 수학 함수들을 포함하는 표준 라이브러리

int main( void )

{

double PI = 3.1415926535;

double x, y;

x = PI / 2.0;

y = sin( x ); // x in radian unit

printf( "sin( %lf ) = %lf\n", x, y );

y = sinh( x );

printf( "sinh( %lf ) = %lf\n", x, y );

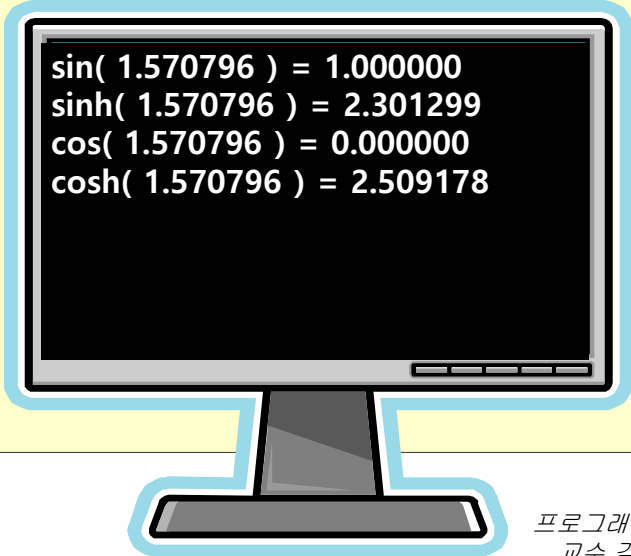
y = cos( x );

printf( "cos( %lf ) = %lf\n", x, y );

y = cosh( x );

printf( "cosh( %lf ) = %lf\n", x, y );

}



```
sin( 1.570796 ) = 1.000000
sinh( 1.570796 ) = 2.301299
cos( 1.570796 ) = 0.000000
cosh( 1.570796 ) = 2.509178
```





# 예제

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

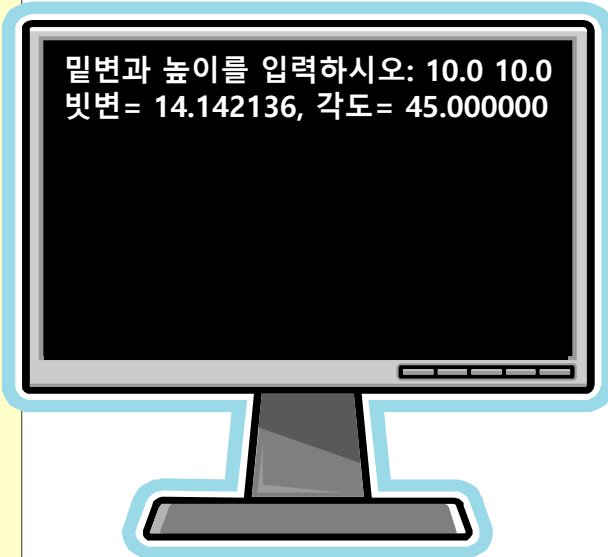
```
#define PI 3.141592653589793
#define RADIANT_TO_DEG (180.0 / (double)PI)
```

```
int main(void)
{
    double base, height, hypotenuse, theta;

    printf("밑변과 높이를 입력하시오:");
    scanf("%lf %lf", &base, &height);

    hypotenuse = sqrt(base * base + height * height);
    theta = RADIANT_TO_DEG * atan2(height, base);
    printf("빗변 = %lf, 각도 = %lf\n", hypotenuse, theta);
    return 0;
}
```

상수를 정의하는 전처리 명령문



밑변과 높이를 입력하시오: 10.0 10.0  
빗변 = 14.142136, 각도 = 45.000000



**지역변수 (local variable),  
전역변수 (global variable),  
정적변수 (static variable),  
함수의 인수 (argument),  
전역변수의 외부 연결 (extern linkage)**

# 전역 변수 (Global Variable) 와 지역 변수 (Local Variable)

```
#include <math.h>
double sumup(int x);

int g_count = 100;

int main(void)
{
    double result_sum;
    ....
    result_sum = sumup(g_count);
    ....
}

double sumup(int x)
{
    double sum = 0.0;
    for (int i=1; i <= x; i++)
    {
        sum = sum + i;
    }
    return sum;
}
```

## 전역 변수 (global variable):

- 함수의 외부에서 정의
- 전역 변수 선언 이후 모든 함수에서 사용 가능

## 지역 변수 (local variable):

- 함수 내부에서 정의
- 지역 변수 선언 이후 해당 함수/블록 내부에서만 사용 가능
- 함수가 종료되면 지역변수는 소멸됨



# 다른 블록에 있는 이름이 같은 지역 변수

```
#include <math.h>
double sum_sqrt(int x);

int main(void)
{
    int count;
    double d;
    . . . . .

    d = sum_sqrt(count);
    . . . . .
}

double sum_sqrt(int x)
{
    int count = x;
    double sq_sum = 0.0;

    for (int i=0; i<count; i++)
    {
        sq_sum = sq_sum + sqrt(i);
    }
    return sq_sum;
}
```

블록이 다를 경우,  
같은 이름의 변수를  
사용할 수 있습니다.



# 지역 변수 예제

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

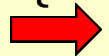
```
int main(void)
```

```
{
```

```
    int i;
```

```
    for(i = 0; i < 5; i++)
```

```
    {
```



```
        int temp = 1;
```

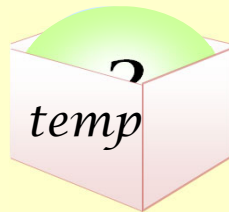
```
        printf("temp = %d\n", temp);
```

```
        temp++;
```

```
    }
```

```
    return 0;
```

```
}
```



블록이 시작할 때 마다  
생성되어 초기화된다.

```
temp = 1  
temp = 1  
temp = 1  
temp = 1  
temp = 1
```



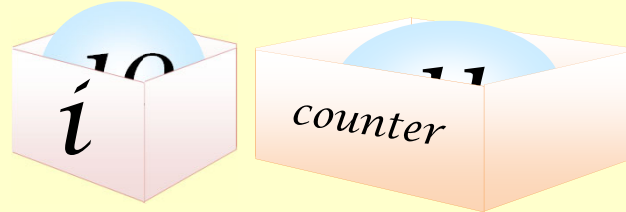
# 함수의 인수 (argument)

```
#include <stdio.h>
int inc(int counter)
```

10

```
int main(void)
{
    int i;

    i = 10;
    printf("함수 호출 전 i=%d\n", i);
    incr(i);
    printf("함수 호출 후 i=%d\n", i);
    return 0;
}
```



함수의 인수도  
일종의 지역변수

```
int incr(int counter)
{
    counter++;
    return counter;
}
```

함수 호출 전 i=10  
함수 호출 후 i=10



# 전역 변수 (Global Variable)

- ◆ 전역 변수(global variable)는 함수 외부에서 선언되는 변수이다.
- ◆ 전역 변수의 범위는 선언된 위치로부터 그 소스 파일의 끝까지 이다.

```
int x = 123;  
void sub1()  
{  
    x = 456;  
    y = 123; // compile error (not defined)  
}  
  
int y = 567;  
void sub2()  
{  
    x = 789;  
    y = 123;  
}
```

**전역 변수 (global variable)**

- 선언된 위치로부터 그 소스 파일의 끝 부분까지의 구간에서 사용가능 함
- 선언된 위치 이전에서는 사용 할 수 없음



# 전역 변수의 초기값과 생존 기간

```
#include <stdio.h>
int counter = 0; // 전역 변수

void set_counter(int i)
{
    counter = i;    // 직접 사용 가능
}

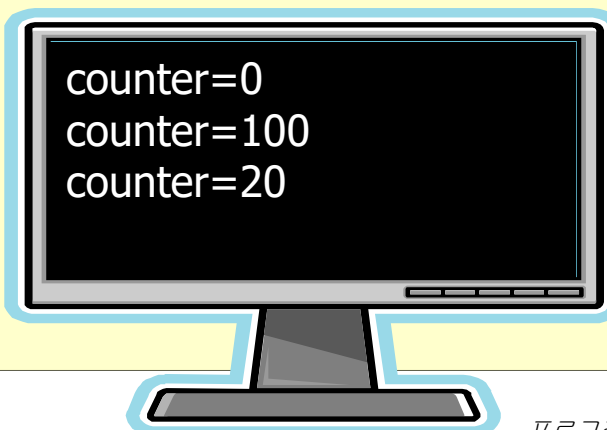
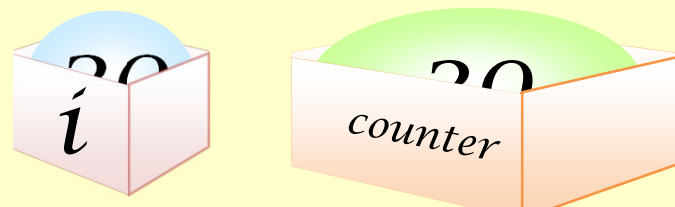
int main(void)
{
    printf("counter=%d\n", counter);

    counter = 100;    // 직접 사용 가능
    printf("counter=%d\n", counter);

    set_counter(20);
    printf("counter=%d\n", counter);

    return 0;
}
```

\*전역변수의 초기값 : 0  
\*생존기간 : 프로그램 시작부터 종료





## 전역 변수의 사용

- ◆ 거의 모든 함수에서 사용하는 공통적인 데이터는 전역 변수로 한다.
- ◆ 일부의 함수들만 사용하는 데이터는 전역 변수로 하지 않고, 지역 변수로 설정하여 사용한다.
- ◆ 전역변수의 경우, 그 값이 다양한 함수에서 변경될 수 있어 올바른 값의 범위를 유지하는 데에 어려움이 있을 수 있고, 디버깅이 매우 어려우므로, 가급적 사용하지 않도록 한다 !!



# 같은 이름의 전역 변수와 지역 변수

```
// 동일한 이름의 전역 변수와 지역 변수
#include <stdio.h>

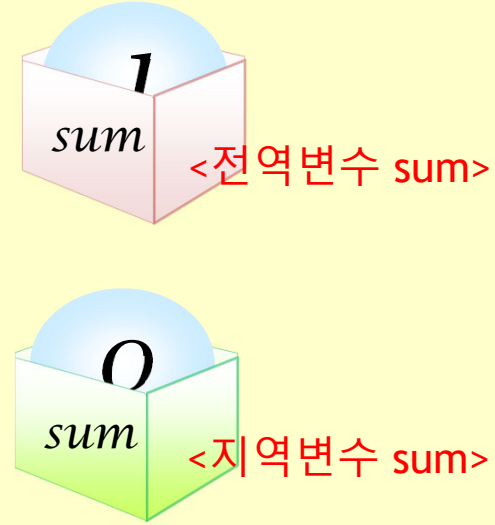
→ int sum = 1;    // 전역 변수

int main(void)
{
    → int sum = 0; // 지역 변수

    printf("sum = %d\n", sum);

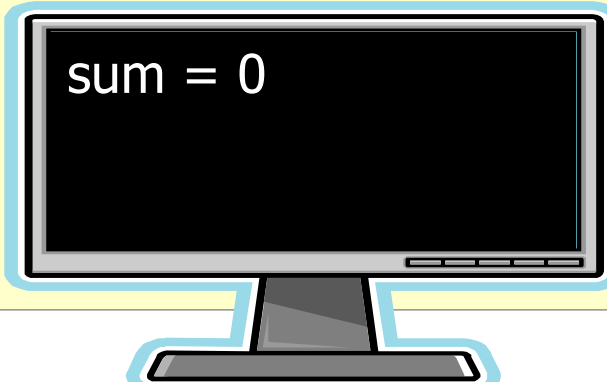
    return 0;
}
```

지역 변수가 전역변수를 가린다.



<전역변수 sum>

<지역변수 sum>



# 정적 할당과 자동할당

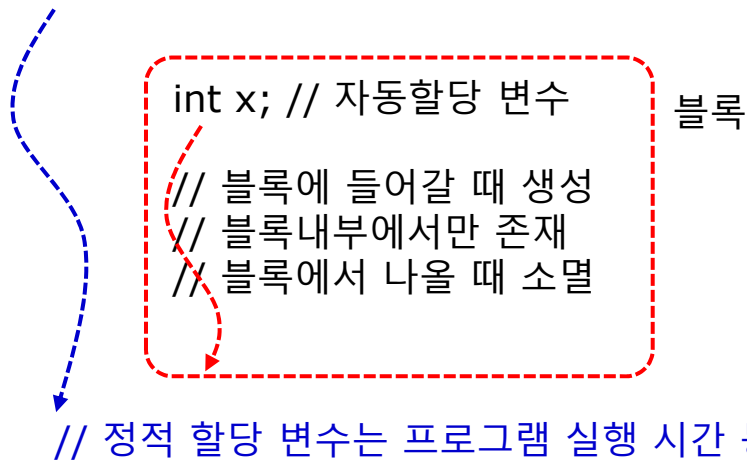
## ◆ 정적 할당(static allocation):

- 프로그램 실행 시간 동안 계속 유지

## ◆ 자동 할당(automatic allocation):

- 블록에 들어갈 때 생성
- 블록에서 나올 때 소멸

static int count; // 정적 할당 변수



# static 저장 유형 지정자

```
#include <stdio.h>
void sub(void);
```

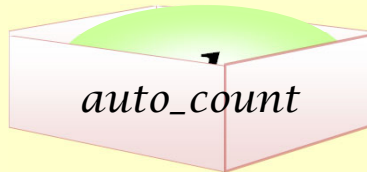
```
int main(void)
{
```

```
    → int i;
      for(i = 0; i < 3; i++)
        sub();
      return 0;
}
```

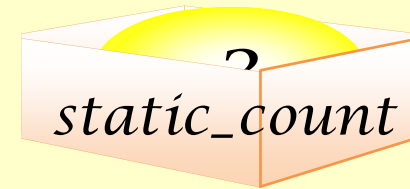
```
void sub(void)
{
```

```
    → int auto_count = 0;
      static int static_count = 0;
```

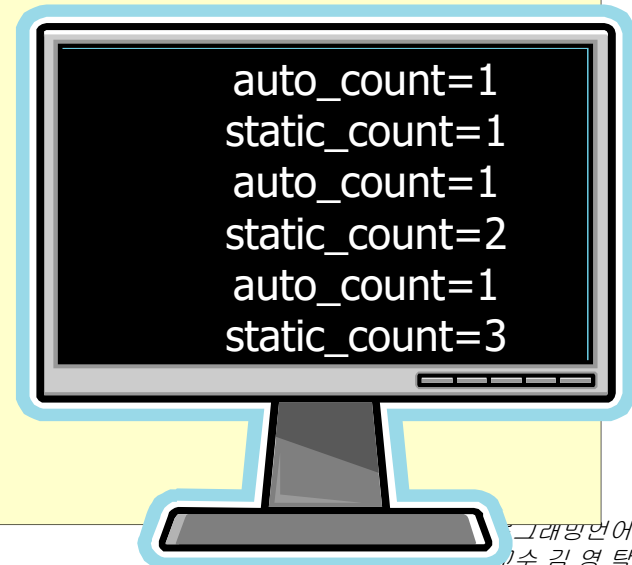
```
    auto_count++;
    static_count++;
    printf("auto_count=%d\n", auto_count);
    printf("static_count=%d\n", static_count);
}
```



자동 지역 변수



정적 지역 변수로써  
static을 붙이면 지역변수가  
정적변수로 된다.



# extern 저장 유형 지정자

## extern1.c

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

```
int x;
```

// 전역 변수

```
extern int y;
```

// 현재 소스 파일의 뒷부분에 선언된 변수

```
extern int z;
```

// 다른 소스 파일의 변수

```
int main(void)
```

```
{
```

```
    extern int x; // 전역 변수 x를 참조한다. 없어도 된다.
```

```
    x = 10;
```

```
    y = 20;
```

```
    z = 30;
```

```
    return 0;
```

```
}
```

```
int y;
```

// 전역 변수

컴파일러에게 변수가 다른 곳  
(다른 소스파일)에서 선언되었음을  
알린다.

## extern2.c

```
int z;
```

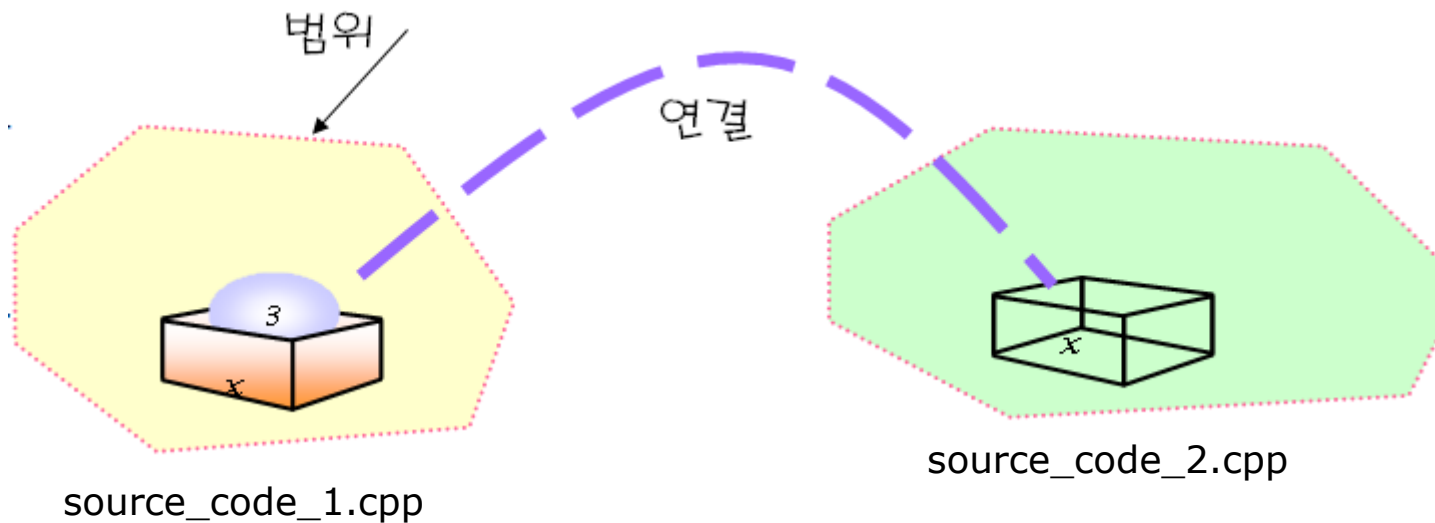


# 연결 (Linkage)

◆ **연결(linkage):** 다른 범위에 속하는 변수들을 서로 연결하는 것

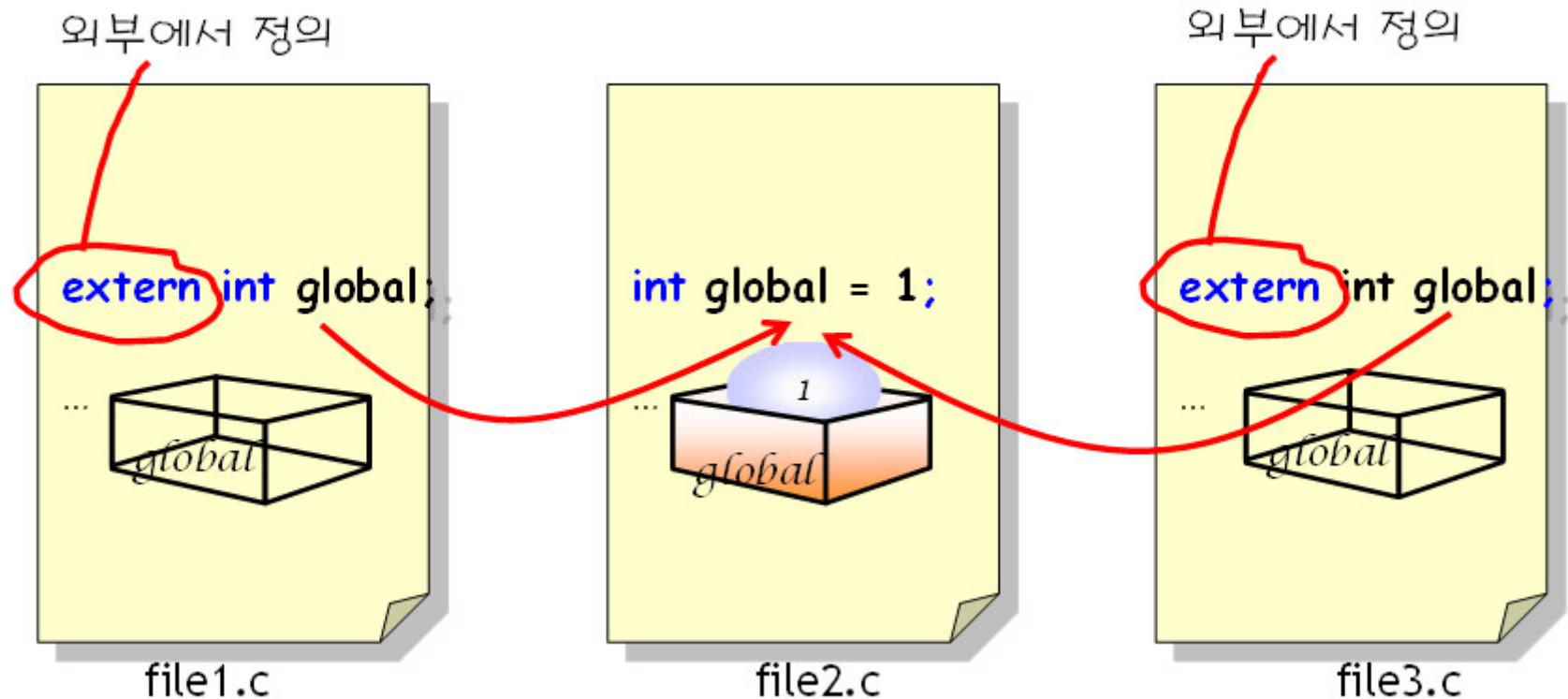
- 외부 연결
- 내부 연결

◆ 전역 변수만이 연결을 가질 수 있다.



# 외부 연결 (External Linkage)

## ◆ 전역 변수를 **extern**을 이용하여서 서로 연결



# 연결 예제

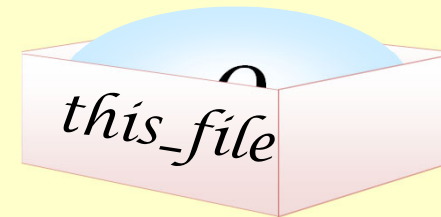
file1.c

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

```
int all_files; // 다른 소스 파일에서도 사용할 수 있는 전역 변수  
static int this_file; // 현재의 소스 파일에서만 사용할 수 있는 정적 전역 변수  
extern void sub();
```

```
int main(void)  
{  
    static int this_function; // 이 함수에서만 사용할 수 있는 정적 지역 변수  
    sub();  
    printf("%d\n", all_files);  
    return 0;  
}
```

연결 (link)



file2.c

```
extern int all_files;  
void sub(void)  
{  
    all_files = 10;  
}
```





**재귀 함수 (Recursive Function)**  
**동적 프로그래밍 (Dynamic Programming)**

# 재귀(recursion) 함수

## ◆ 재귀함수 호출 (recursive function call) 이란?

- 알고리즘이나 함수가 수행 도중에 자기 자신을 다시 호출하여 문제를 해결하는 기법

## ◆ Factorial의 정의

$$n! = \begin{cases} 1 & n = 1 \\ n \times (n - 1)! & n \geq 2 \end{cases}$$

## ◆ power의 정의

$$power(x, y) = \begin{cases} 1 & y = 0 \\ x \times power(x, y - 1) & y > 0 \end{cases}$$

## ◆ Fibonacci 수열의 정의

$$fibo(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fibo(n - 2) + fibo(n - 1) & n \geq 2 \end{cases}$$



# 재귀 알고리즘의 구조

## ◆ 재귀 알고리즘은 다음과 같은 부분들을 포함한다.

- 재귀 호출을 하는 부분
- 재귀 호출을 멈추는 부분

```
int factorial(int n)
{
    if( n == 1 )
        return 1;
    else
        return n * factorial(n-1);
}
```

← 재귀를 멈추는 부분

← 재귀호출을 하는 부분

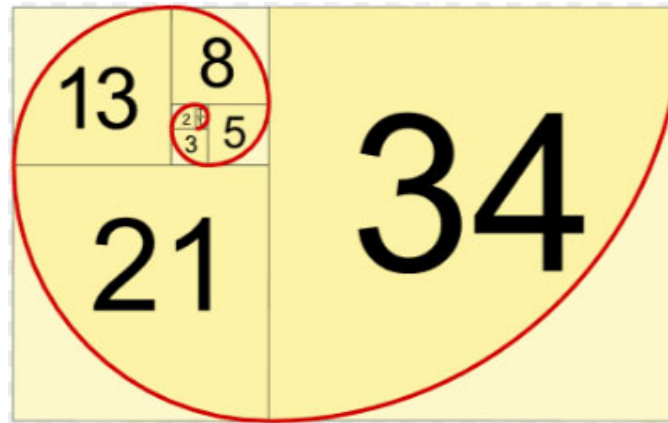
## ◆ 만약 재귀 호출을 멈추는 부분이 없다면?.

- 시스템 오류가 발생할 때까지 무한정 호출하게 된다.



# 피보나치 수열

## ◆ Fibonacci Series



# 피보나치 수열 (Fibonacci Series) 계산

◆ 순환 호출을 사용하면 비효율적인 예

◆ 피보나치 수열

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

$$fib(n) \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-2) + fib(n-1) & otherwise \end{cases}$$

◆ 순환적인 구현

```
int fibo(int n)
{
    if( ( n==0 ) || ( n==1 ) )
        return n;
    return (fibo(n-1) + fibo(n-2));
}
```



# time()을 사용한 함수 실행 시간 측정

## double Fibo(int n)

```
{
    double f1, f2;

    if ((n == 0) || (n == 1))
        return double(n);
    else
    {
        f1 = Fibo(n - 1);
        f2 = Fibo(n - 2);
        return ( f1 + f2 );
    }
}
```

```
/* main() for RecursiveFunctions */
```

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

```
#include <time.h>
```

```
double Fibo(int n);
```

```
void main()
```

```
{
```

```
    time_t t_before, t_after;
```

```
    double t_diff;
```

```
    double fibo_res;
```

```
    /* Testing Fibonacci */
```

```
    for (int i = 0; i < 10; i++)
```

```
    {
```

```
        printf("Fibonacci(%2d) = %15.2lf\n", i, Fibonacci(i));
```

```
    }
```

```
    for (int i = 10; i <= 45; i += 5)
```

```
    {
```

```
        time(&t_before);
```

```
        fibo_res = Fibo(i);
```

```
        time(&t_after);
```

```
        printf("Fibonacci(%2d) = %15.2lf, ", i, fibo_res);
```

```
        t_diff = difftime(t_after, t_before);
```

```
        printf(" ==> processing time %7.2lf seconds\n",
```

```
            t_diff);
```

```
    }
```

```
}
```



## 실행결과

---

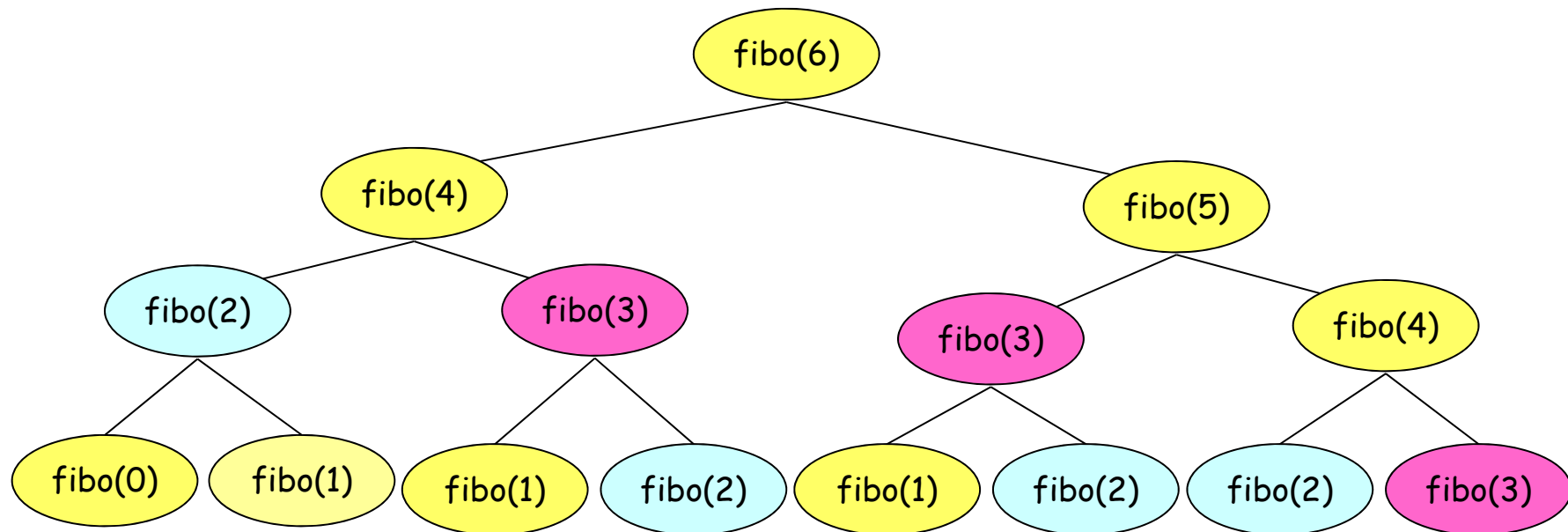
```
Fibonacci( 0) =          0.00
Fibonacci( 1) =          1.00
Fibonacci( 2) =          1.00
Fibonacci( 3) =          2.00
Fibonacci( 4) =          3.00
Fibonacci( 5) =          5.00
Fibonacci( 6) =          8.00
Fibonacci( 7) =         13.00
Fibonacci( 8) =         21.00
Fibonacci( 9) =         34.00
Fibonacci(10) =         55.00, ==> processing time 0.00 seconds
Fibonacci(15) =        610.00, ==> processing time 0.00 seconds
Fibonacci(20) =       6765.00, ==> processing time 0.00 seconds
Fibonacci(25) =      75025.00, ==> processing time 0.00 seconds
Fibonacci(30) =     832040.00, ==> processing time 0.00 seconds
Fibonacci(35) =    9227465.00, ==> processing time 0.00 seconds
Fibonacci(40) =   102334155.00, ==> processing time 9.00 seconds
Fibonacci(45) =  1134903170.00, ==> processing time 88.00 seconds
계속하려면 아무 키나 누르십시오 . . .
```



# 동적 프로그래밍 (Dynamic Programming)

## ◆ 피보나치 수열계산에서 순환 호출을 사용했을 경우의 비효율성

- 같은 항이 중복해서 계산됨
- 예를 들어 fibo(6)을 호출하게 되면 fibo(3)이 4번이나 중복되어서 계산됨
- 이러한 현상은 n이 커지면 더 심해짐



## ◆ 계산 결과 표를 사용하여 동일한 인수를 사용한 재귀함수 호출이 여러 번 발생하지 않도록 개선





# 동적 프로그래밍 기법을 사용하여 개선된 Fibonacci 수열 계산 함수

```
double dynProg_Fibo(int n)
{
    double fibo_res;
    static double fibo_series[FIBO_TBL_SIZE] = { 0 };

    if ((n == 0) || (n == 1))
    {
        return n;
    }
    else if (fibo_series[n] != 0)
    {
        fibo_res = fibo_series[n];
        return fibo_res;
    }
    else
    {
        fibo_res = dynProg_Fibo(n - 2) + dynProg_Fibo(n - 1);
        fibo_series[n] = fibo_res;
        return fibo_res;
    }
}
```



```
/* main() for RecursiveFunctions */
#include <stdio.h>
#include <time.h>
```

```
double dynProg_Fibo(int n);
```

```
void main()
```

```
{
```

```
    time_t t_before, t_after;
```

```
    double t_diff;
```

```
    double fibo_res;
```

```
    /* Testing Fibonacci */
```

```
    for (int i = 0; i < 10; i++)
```

```
    {
```

```
        printf("Fibonacci(%2d) = %15.2lf\n", i, Fibonacci(i));
```

```
    }
```

```
    for (int i = 10; i <= 45; i += 5)
```

```
    {
```

```
        time(&t_before);
```

```
        //fibo_res = Fibonacci(i);
```

```
        //printf("Fibonacci(%2d) = %15.2lf, ", i, fibo_res);
```

```
        fibo_res = dynProg_Fibo(i);
```

```
        time(&t_after);
```

```
        printf("dynProg_Fibo (%2d) = %15.2lf, ", i, fibo_res);
```

```
        t_diff = difftime(t_after, t_before);
```

```
        printf(" ==> processing time %7.2lf seconds\n", t_diff);
```

```
    }
```

```
}
```

```
Fibonacci( 0) =          0.00
Fibonacci( 1) =          1.00
Fibonacci( 2) =          1.00
Fibonacci( 3) =          2.00
Fibonacci( 4) =          3.00
Fibonacci( 5) =          5.00
Fibonacci( 6) =          8.00
Fibonacci( 7) =         13.00
Fibonacci( 8) =         21.00
Fibonacci( 9) =         34.00
Adv_Fibo (10) =         55.00, ==> processing time 0.00 seconds
Adv_Fibo (15) =        610.00, ==> processing time 0.00 seconds
Adv_Fibo (20) =       6765.00, ==> processing time 0.00 seconds
Adv_Fibo (25) =      75025.00, ==> processing time 0.00 seconds
Adv_Fibo (30) =     832040.00, ==> processing time 0.00 seconds
Adv_Fibo (35) =    9227465.00, ==> processing time 0.00 seconds
Adv_Fibo (40) =   102334155.00, ==> processing time 0.00 seconds
Adv_Fibo (45) =  1134903170.00, ==> processing time 0.00 seconds
계속하려면 아무 키나 누르십시오 . . .
```



## **Homework 4**

# Homework 4

## 4.1 중복되지 않는 big random number array 생성 및 샘플 출력

- 표준 라이브러리 <stdlib.h>에 포함된 rand() 함수는 0 ~ RAND\_MAX (32767) 범위의 난수를 생성하므로, 더 큰 범위의 난수 (예를 들어 0 ~ 100,000)는 사용할 수 없다.
- 이와 같이 RAND\_MAX를 초과하는 범위의 중복되지 않는 난수 (즉, unsigned int가 표현할 수 있는 범위: 0 ~  $2^{30}-1$ )를 발생시킬 수 있는 함수 **genBigRandArray(int \*array, int size)**를 작성하라. (Hint: 15비트 단위의 난수를 발생시키는 rand()함수를 두 번 호출하여, 이 값을 unsigned long의 상위 15비트와 하위 15비트로 각각 사용하는 방법을 고려할 것.)
- 100개 이상의 원소를 가지는 큰 정수 배열에서 첫 부분과 마지막 부분의 샘플을 출력하는 함수 **printArraySample(int \*array, int size, int line\_size)**를 작성하라. 샘플은 한 줄에 line\_size 개씩 출력하며, 첫 부분 3 줄과 마지막 부분 3줄을 출력하도록 할 것. 이 함수 호출에서 line\_size가 설정되어 있지 않는 경우, 기본값으로 10을 사용할 것.
- C 프로그램으로 작성된 genBigRandArray(int \*array, int size) 함수와 printArraySample(int \*array, int size, int line\_size) 함수를 사용하여 size가 200,000 ~ 1,000,000인 큰 난수배열을 생성하고, 생성된 큰 난수 배열의 샘플을 출력하는 기능을 시험하는 main() 함수를 구현하고, 그 결과를 출력하라.



## ◆ main() 프로그램 (예시)

```
/* main() for big random array test */

#include <stdio.h>
#include <stdlib.h>
#define LINE_SIZE 10
void genBigRandArray(int *array, int size);
void printArraySample(int *array, int size, int line_size);
void main()
{
    int* array; // pointer for dynamic array

    for (int size = 200000; size <= 1000000; size += 200000)
    {
        printf("Testing generation of dynamic array of random numbers (size: %d)\n", size);
        array = (int*)calloc(size, sizeof(int)); // dynamic array
        if (array == NULL)
        {
            printf("Error in dynamic memory allocation for integer array of size (%d)\n", size);
            exit(-1);
        }
        genRandArray(array, size);
        printArraySample(array, size, LINE_SIZE);
    }
}

void genBigRandArray(int *array, int size);
{ . . . . }
void printArraySample(int *array, int size, int line_size)
{ . . . . }
```



## ◆ 실행결과 (예시)

```

Testing generation of dynamic array of random numbers (size: 200000)
79934 65861 138171 11708 158729 40390 175043 120171 94543 120553
140102 199164 171542 186141 178661 112329 106103 190339 174482 23708
174240 193778 100116 137399 123216 123315 100423 14349 99295 75142

39697 83544 11166 44100 110670 116935 114294 82319 32994 20253
36404 190502 63953 125983 22434 58138 148825 91261 16620 114859
112553 56072 142812 99537 9359 100105 32120 197945 68562 126196

Testing generation of dynamic array of random numbers (size: 400000)
79934 65861 138171 211708 358729 240390 375043 120171 94543 120553
340102 399164 371542 186141 178661 312329 306103 390339 374482 23708
174240 193778 100116 337399 323216 123315 300423 214349 99295 75142

395023 354231 102165 257783 288735 349485 191397 173492 152576 290228
150384 266720 236779 188004 42335 189393 381964 1579 381294 85053
154268 365501 360561 179158 57098 79884 147245 326196 124588 14348

Testing generation of dynamic array of random numbers (size: 600000)
188987 457476 2193 182797 585029 390770 67749 383141 110600 109518
169535 410339 493705 248569 449670 437465 590062 23831 126403 277407
393732 154257 229924 357007 416827 149524 397050 441991 340452 544719

135601 54150 549665 105555 202866 546745 216536 244026 513451 316932
241269 203044 496028 547696 176403 389614 415394 326440 331869 521216
92622 113274 402052 568906 173908 139232 121748 501810 253023 108610

Testing generation of dynamic array of random numbers (size: 800000)
330807 674499 375272 586653 735041 541150 335047 755167 559425 498483
322680 628873 41276 518356 146087 562602 332198 199146 711092 531106
213225 147504 152372 776615 685029 284788 384622 269633 214378 105241

55007 174508 107412 681199 67986 399418 761202 70279 592346 154887
478539 47278 210855 207658 751192 478689 87777 50968 97405 522858
604595 9053 654979 408130 594198 633270 49471 223450 250165 604801

Testing generation of dynamic array of random numbers (size: 1000000)
439860 33347 181119 757742 394109 691530 827754 218137 775483 945623
352113 305584 163439 380784 617096 120507 648925 465406 863014 352037
32718 507983 849412 963455 378639 878228 914017 97275 688304 632995

147684 376481 471087 575464 486924 26725 474251 458851 769638 751518
75160 253846 855304 699811 194904 778954 642895 496528 265012 99189
78311 726613 315628 706291 882024 635827 633580 687387 68411 876878

```



## 4.2 power(base, exponent) 함수

double power(double base, int exponent) 함수는  $\text{base}^{\text{exponent}}$  거듭제곱 (지수 승)을 계산하는 함수이다. 이 거듭제곱은 재귀함수 구조 또는 반복문 구조로 구현할 수 있다.

power(base, exponent) 를 재귀 (recursive) 구조와 반복 (iterative) 구조로 각각 작성하라.

```
double powerIter(double base, int exponent); // 반복문 구조
```

```
double powerRecur(double base, int exponent); // 재귀함수 구조
```

main() 함수에서는 이 두 거듭제곱 함수를 차례로 호출하며 base, exponent를 인수 (argument로) 전달하며, exponent값이 1000, 2000, 3000, 4000 인 경우에 대하여 각각 실행하라.

```
double base = 1.015
```

```
for (int exponent=1000; exponent<=4000; exponent+=1000)
```

```
{
```

```
    result_i = powerIter(base, exponent);
```

```
    result_r = powerRecur(base, exponent);
```

```
}
```

각 함수 실행에서 걸린 시간을 다음 main() 예제 함수와 같이 초 단위의 시간측정이 가능한 time() 함수와 마이크로 초 단위의 시간 측정이 가능한 QueryPerformanceCounter() 함수를 사용하여 측정하라.

측정된 시간이 함수 구현 방법에 따라 차이가 나는 이유를 설명하라.



# main()

```
/* main_recursive_vs_iterative.c (1) */

#include <stdio.h>
#include <time.h>
#include <Windows.h>

double powerRecur(double base, int exponent);
double powerIter(double base, int exponent);

void main()
{
    time_t t_before, t_after;
    int t_diff;
    LONGLONG t_diff_pc;
    LARGE_INTEGER freq, t1, t2;
    double t_elapse_ms;

    double result_i, result_r;

    double base = 1.015;
```





```
/* main_recursive_vs_iterative.c (2) */
```

```
QueryPerformanceFrequency(&freq);
```

```
for (int expo = 1000; expo <= 4000; expo += 1000)
```

```
{
```

```
    time(&t_before);
```

```
    QueryPerformanceCounter(&t1);
```

```
    result_i = powerIter(base, expo);
```

```
    QueryPerformanceCounter(&t2);
```

```
    time(&t_after);
```

```
    t_diff = difftime(t_after, t_before);
```

```
    t_diff_pc = t2.QuadPart - t1.QuadPart;
```

```
    t_elapse_ms = ((double)t_diff_pc / (double)freq.QuadPart) * 1000;
```

```
    printf("PowerIter(1.015, %d) by iterative = %40.5lf, took (%3d) sec,  
          (%10.2lf) milli-second\n", expo, result_i, t_diff, t_elapse_ms);
```

```
    time(&t_before);
```

```
    QueryPerformanceCounter(&t1);
```

```
    result_r = powerRecur(base, expo);
```

```
    QueryPerformanceCounter(&t2);
```

```
    time(&t_after);
```

```
    t_diff = difftime(t_after, t_before);
```

```
    t_diff_pc = t2.QuadPart - t1.QuadPart;
```

```
    t_elapse_ms = ((double)t_diff_pc / freq.QuadPart) * 1000;
```

```
    printf("PowerRecur(1.015, %d) by recursive = %40.5lf, took (%3d) sec,  
          (%10.2lf) milli-second\n", expo, result_r, t_diff, t_elapse_ms);
```

```
    } // end for
```

```
}
```

```
double powerRecur(double base, int exponent) { . . . }
```

```
double powerIter(double base, int exponent) { . . . }
```



## ◆ 실행결과

```
PowerItera(1.015, 1000) by iterative =          2924436.86039, took ( 0) sec, (    2.82) micro-second
PowerRecur(1.015, 1000) by recursive =          2924436.86039, took ( 0) sec, (   95.71) micro-second
PowerItera(1.015, 2000) by iterative =          8552330950415.20020, took ( 0) sec, (    5.65) micro-second
PowerRecur(1.015, 2000) by recursive =          8552330950415.20020, took ( 0) sec, (  137.79) micro-second
PowerItera(1.015, 3000) by iterative =         25010751873659445000.00000, took ( 0) sec, (    8.21) micro-second
PowerRecur(1.015, 3000) by recursive =         25010751873659445000.00000, took ( 0) sec, (  158.32) micro-second
PowerItera(1.015, 4000) by iterative =       731423646854299550000000000.00000, took ( 0) sec, (   11.80) micro-second
PowerRecur(1.015, 4000) by recursive =       731423646854299550000000000.00000, took ( 0) sec, (  215.79) micro-second
계속하려면 아무 키나 누르십시오 . . .
```

