C Programming – Day 2

JunGu Kang
Dept. of Cyber Security

아주대학교 Whois

함수

함수

- 큰 작업을 작은 작업들로 나눈다.
- 누군가 만들어 둔 것을 가져다 쓰자.
- 굳이 알 필요가 없는 것들을 숨겨준다.
  - 코드를 전체적으로 깔끔하게 만들어준다.
- 코드를 수정하기 편해진다.

함수

You don’t have to reinvent the wheel,
just attach it to a new wagon.

- Mark McCormack

함수

이미 많은 함수가 만들어져 있으니 잘 써먹자.

함수의 정의

그럼에도 불구하고 내가 원하는 함수를 만들고 싶다면?

함수의 정의

```
return-type function-name(argument declarations) {
    declarations and statements
}
```

함수의 정의

```
return-type function-name(arguments) {
    declarations and statements
    return expression;
}
```

함수의 정의

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

함수의 정의

```
double add_double(double a, double b) {
    return a + b;
}
```

함수의 정의

```
void print_int(int a) {
    printf(“%d”, a);
}
```

함수의 정의

```
int return_true() {
    return 1;
}
```

함수의 정의

```
int main(void) {
    statements
    return 0;
}
```

함수의 정의

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

main() {
    add(2, 3);
    return 0;
}
```

함수의 정의

```
int main(void) {
    add(2, 3);  // add함수가 선언되어있지 않기 때문에 오류가 발생한다.
    return 0;
}

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

Prototype

```
int add(int a, int b);  // 함수를 선언해주면 된다

int main(void) {
    add(2, 3);
    return 0;
}

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

Prototype

```
int add(int, int);  // 변수의 이름은 생략할 수 있다

int main(void) {
    add(2, 3);
    return 0;
}

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

헤더 파일

라이브러리 함수도 사용하려면 선언해야 한다.

헤더 파일

printf함수는 선언하지 않고 사용했는데?

헤더 파일

헤더파일에 선언되어 있기 때문이다.

헤더 파일

라이브러리 함수를 쓰기 위해서는
라이브러리 함수가 선언되어 있는 헤더 파일을 추가해야 한다.

헤더 파일

```
#include <filename>
```

헤더 파일

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

Static Function

```
static int add(int a, int b) {
    return a + b;
}  // 이 함수는 이 소스코드에서만 사용할 수 있다

int main(void) {
    add(2, 3);
    return 0;
}
```

변수

변수

- Automatic Variable
- Global Variable
- External Variable
- Static Variable
- Register Variable
- Volatile Variable

Automatic Variable

```
change() {
    a += 5;
}

main() {
    int a;
    a = 10;

    change();
}
```

Static Variable

Automatic Variable은
함수가 호출될 때 할당되고, 함수가 종료될 때 소멸된다.

Automatic Variable

```
c() {
    int c = 15;
}

b() {
    int b = 5;
    b += 5;
    c();
}

main() {
    int a = 10;
    b();
}
```

Low
High

Automatic Variable

```
c() {
    int c = 15;
}

b() {
    int b = 5;
    b += 5;
    c();
}

main() {
    int a = 10;
    b();
}
```

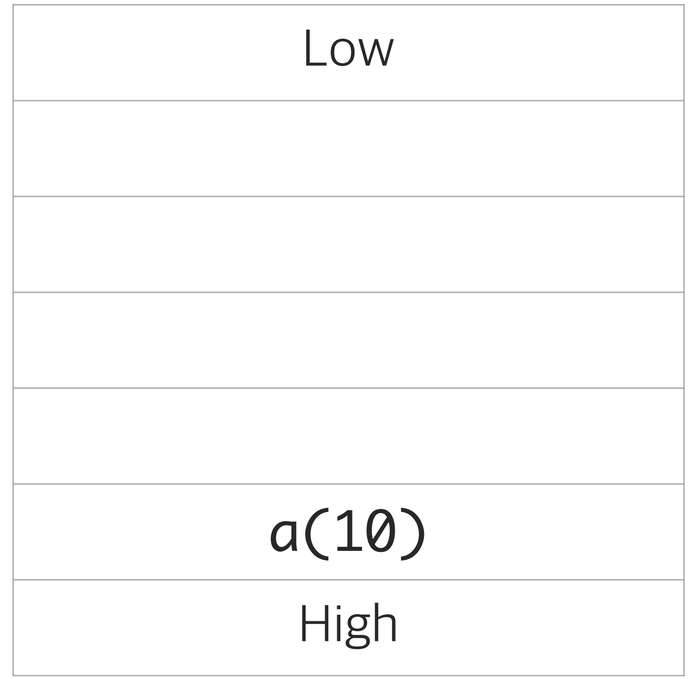
Low
a(10)
High

Automatic Variable

```
c() {
    int c = 15;
}

b() {
    int b = 5;
    b += 5;
    c();
}

main() {
    int a = 10;
    b();
}
```

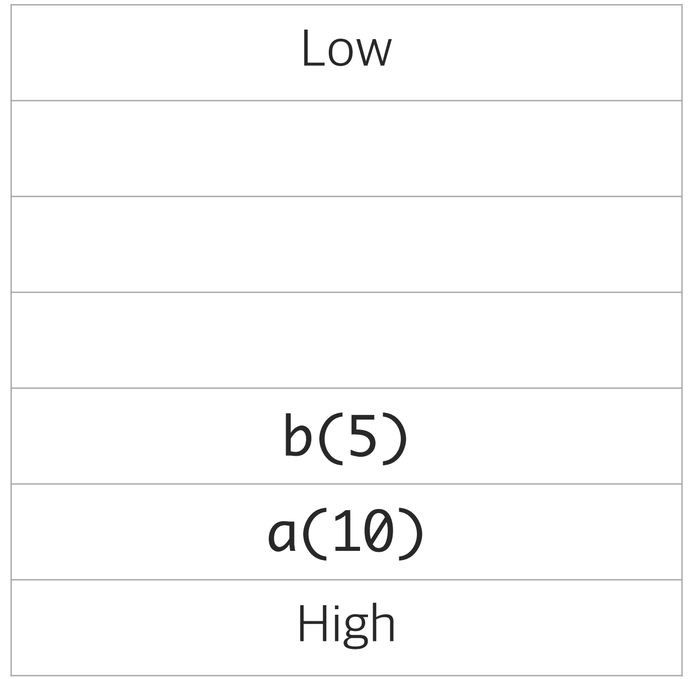


Automatic Variable

```
c() {
    int c = 15;
}

b() {
    int b = 5;
    b += 5;
    c();
}

main() {
    int a = 10;
    b();
}
```



Automatic Variable

```
c() {
    int c = 15;
}

b() {
    int b = 5;
    b += 5;
    c();
}

main() {
    int a = 10;
    b();
}
```

Low
b(10)
a(10)
High

Automatic Variable

```
c() {
    int c = 15;
}

b() {
    int b = 5;
    b += 5;
    c();
}

main() {
    int a = 10;
    b();
}
```

Low
b(10)
a(10)
High

Automatic Variable

```
c() {
    int c = 15;
}

b() {
    int b = 5;
    b += 5;
    c();
}

main() {
    int a = 10;
    b();
}
```

Low
c(15)
b(10)
a(10)
High

Automatic Variable

```
c() {
    int c = 15;
}

b() {
    int b = 5;
    b += 5;
    c();
}

main() {
    int a = 10;
    b();
}
```

Low
b(10)
a(10)
High

Automatic Variable

```
c() {
    int c = 15;
}

b() {
    int b = 5;
    b += 5;
    c();
}

main() {
    int a = 10;
    b();
}
```

Low
b(10)
a(10)
High

Automatic Variable

```
c() {
    int c = 15;
}

b() {
    int b = 5;
    b += 5;
    c();
}

main() {
    int a = 10;
    b();
}
```

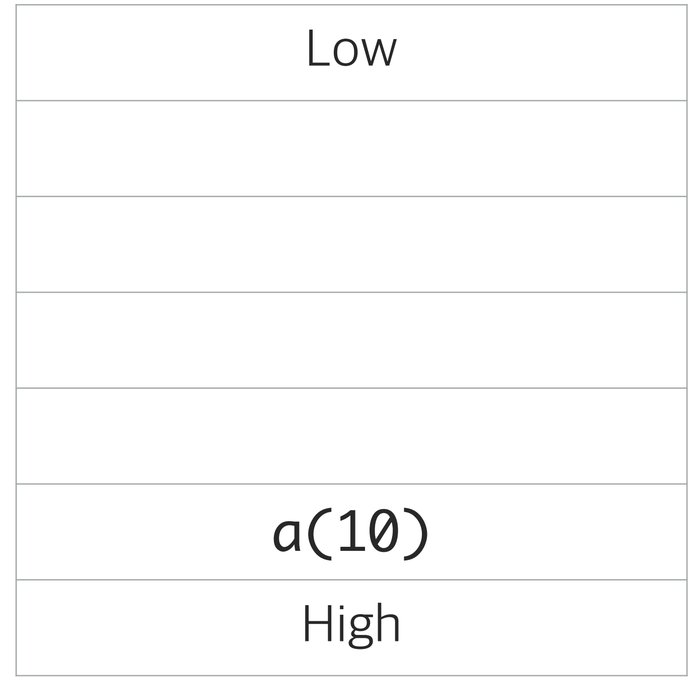
Low
a(10)
High

Automatic Variable

```
c() {
    int c = 15;
}

b() {
    int b = 5;
    b += 5;
    c();
}

main() {
    int a = 10;
    b();
}
```



Automatic Variable

```
c() {
    int c = 15;
}

b() {
    int b = 5;
    b += 5;
    c();
}

main() {
    int a = 10;
    b();
}
```

Global Variable

```
func_one() {
    a += 5;
}

func_two() {
    a += 5;
}

a = 5;

main() {
    func_one();
    func_two();
}
```

전역변수를 남용하지 말자

반드시 전역변수가 필요한 상황이 아니라면
지역변수를 사용한다.

External Variable

```
extern int a;  // 다른 소스코드에서 선언한 변수 a를 이 소스코드에서 사용

main() {
    printf(“%d”, a);
}
```

Static Variable(Local)

```
func1() {
    static int a = 5;  // 이 변수는 이 함수에서만 사용 가능
    a += 5;
}

func2() {
    a += 5;  // a가 선언되지 않아 오류 발생
}

main() {
    func1();
    func2();
}
```

Static Variable(Global)

```
static int a = 5;  // 이 변수는 이 소스코드에서만 사용할 수 있다.
                   // 다른 소스코드에서 외부변수로 쓸 수 없다.

main() {
    statements
}
```

Static Variable

Static Variable은 Global Variable처럼
프로그램이 시작될 때 할당되고,
프로그램이 종료될 때 소멸된다.

Static Variable

가능하다면 Global Variable을 Static Variable로 대체하자

Register Variable

```
main() {
    register int a = 5;  // 이 변수는 register에 저장하는 것이 좋다고 알림
}
```

Register Variable

register 키워드를 붙인다고 해서
항상 register에 저장되지는 않는다.

Volatile Variable

```
main() {
    volatile int a = 5
}
```

Volatile Variable

이 변수를 최적화하지 않는다.

Volatile Variable

- 레지스터가 아닌 메모리에 저장된다.
- 항상 직접 그 값을 읽어서 확인한다.

Volatile Variable

```
main() {
    int a = 10;

    while(a == 10) {  // a를 누군가가 바꾸지 않는 한 계속 10이다.
        statements
    }
}
```

Volatile Variable

```
main() {
    int a = 10;

    while(1) {  // 항상 참이므로 이렇게 최적화할 수 있다.
        statements
    }
}
```

Volatile Variable

당연히 이렇게 최적화 하는 것이 옳지만,
이렇게 최적화해서는 안되는 경우도 있다.

Shadowing

```
int a = 5;

main() {
    int a = 10;  // this automatic variable a shadows global variable a

    printf("%d", a);  // 전역변수가 아닌 지역변수 a의 값을 출력한다
}
```

재귀

Recursion

함수가 **자기 자신을 다시 호출**한다.

Recursion

```
int add(int n) {
    if(n == 1) return 1;
    else return n + add(n - 1);
}

main() {
    printf("%d", add(10));
}
```

배열

Array

변수들의 집합

배열의 선언

```
type name[length];
```

배열의 선언

```
int student_no[100];
char string[1000];
```

배열의 선언

```
int length = 100;
char string[length];  // 이렇게 선언하면 안된다.
```

배열의 선언

배열의 길이는 상수로만 선언할 수 있다.

* 최신 표준에서는 변수로도 선언할 수 있으나 이 강의는 ANSI C를 기준으로 한다.

배열의 선언과 초기화

```
int array1[5] = {1, 2, 3, 4, 5};
int array2[5] = {1, 2, 3};  // 이렇게 생략하면 나머지는 0으로 초기화된다.
int array3[5] = {0};  // 그래서 이렇게 쓰면 모두 0으로 초기화된다.
int array4[] = {1, 2, 3, 4, 5, 6, 7, 8};  // 길이는 자동으로 8이 된다.
```

배열 원소에 접근

```
main() {
    int arr[5];

    arr[0] = 1, arr[1] = 2, arr[2] = 3, arr[3] = 4, arr[4] = 5;

    printf("%d, %d, %d, %d, %d\n", arr[0], arr[1], arr[2], arr[3], arr[4]);
}
```

배열 원소에 접근

```
main() {
    int arr[5];

    arr[0] = 1, arr[1] = 2, arr[2] = 3, arr[3] = 4, arr[4] = 5;

    for(int i = 0; i < 5; i++) printf("%d, ", arr[i]);
    printf("\n");
}
```

배열의 원소에 접근

```
main() {
    int arr[5];

    printf("%d", arr[-1]);  // 문제없이 접근 가능하다.
    printf("%d", arr[100]);  // 이것도 마찬가지.
}
```

배열의 크기와 길이

```
main() {
    int arr[5];

    printf(“size: %d\n”, sizeof arr);  // 크기
    printf(“length: %d\n”, sizeof arr / sizeof (int));  // 길이
}
```

배열과 문자열

문자열은 문자들의 Sequence이다.
즉, char형 변수들을 이어붙인 배열이다.

배열과 문자열

```
char string1[10] = “Hello!”;  // 문자열은 이렇게 초기화할 수 있다.
char string2[] = “Hello, World!”;  // 문자열의 경우에도 자동으로 14가 된다.
```

배열과 문자열

문자열은 Null로 끝난다.

‘H’	‘e’	‘l’	‘l’	‘o’	\0

배열과 문자열

실제로는 이렇게 저장되어 있을 수 있다.

‘H’	‘e’	‘l’	‘l’	‘o’	\0	27	34	75	25	47	89	38	42	86	24	47

배열과 문자열

문자열의 끝에 Null이 없다면?

‘H’	‘e’	‘l’	‘l’	‘o’	37	27	34	75	25	47	89	38	42	86	24	47

배열과 문자열

```
main() {
    int string[100];

    scanf("%s", string);  // & 연산자가 붙지 않는다.
}
```

배열과 문자열

```
main() {
     int string[100];

     printf(“%s”, string);
}
```

다차원 배열

변수를 꼭 한 줄로만 이어붙여야 하는가?

다차원 배열의 선언과 초기화

```
int 2d_array[2][2] = { {1, 2}, {3, 4} };
int 3d_array[2][2][2] = { { {1, 2}, {3, 4}, },
                          { {5, 6}, {7, 8}, } };
int 4d_array[2][2][2][2] = { { { {1, 2}, {3, 4} },
                               { {5, 6}, {7, 8} } },
                             { { {9, 10}, {11, 12} },
                               { {13, 14}, {15, 16} } };
                            // 이런짓은 되도록 하지 말자.
```

다차원 배열

배열의 차원에는 제한이 없지만 4차원 이상은 쓰지 말자.

다차원 배열의 원소에 접근

```
main() {
    int 2d_array[2][2] = { {1, 2}, {3, 4} };

    printf("%d, %d\n", 2d_array[0][0], 2d_array[0][1]);
    printf("%d, %d\n", 2d_array[1][0], 2d_array[1][1]);
}
```

다차원 배열의 원소에 접근

```
main() {
    int 2d_array[10][10];

    for(int i = 0; i < 10; i++) {
        for(int j = 0; j < 10; j++) printf("%d ", 2d_array[i][j]);
        printf("\n");
    }
}
```

포인터

Pointer

변수는 메모리에 할당되고 메모리에는 **주소값**이 있다.

Pointer

메모리의 주소를 가리키는 변수

Pointer

포인터 변수는 메모리의 주소값을 저장한다.

포인터 변수의 선언

```
type * name;
```

포인터 변수의 선언

```
char * char_ptr;
int * int_ptr;
long * long_ptr;
float * float_ptr;
double * double_ptr;
```

포인터 변수의 선언과 초기화

```
char * char_ptr = NULL;
int * int_ptr = NULL;
long * long_ptr = NULL;
float * float_ptr = NULL;
double * double_ptr = NULL;
```

포인터의 선언과 초기화

포인터는 꼭 **NULL로 초기화**한다.
초기화하지 않은 상태로 두지 말자.

포인터의 자료형

포인터 변수는 메모리의 주소를 저장하기 때문에
크기가 type과 관계없이 같다.

포인터의 자료형

포인터의 type은 포인터의 크기가 아니라

포인터가 가리키는 변수를 어떻게 읽고 쓸지 결정한다.

포인터 연산자

```
main() {
    int a = 10;
    int * a_ptr = &a;  // a의 주소값을 a_ptr에 저장
    printf("address: %p\n", a_ptr);
    printf("value: %d", * a_ptr);  // a_ptr가 가리키는 메모리 주소에 저장된 값(역참조)
}
```

Pointer

```
main() {
    int a = 10, b = 20;
    int * ptr = NULL;

    ptr = &a;
    printf("address: %p\n", ptr);
    printf("value: %d", * ptr);

    ptr = &b;
    printf("address: %p\n", ptr);
    printf("value: %d", * ptr);
}
```

Pointer

```
main() {
     char * char_ptr = 100;
     // 이렇게 초기화하면 안된다.

     printf(“%p\n”, ++char_ptr);
     printf(“%p\n”, ++char_ptr);
     printf(“%p\n”, ++char_ptr);
}
```

Pointer

```
main() {
    int * int_ptr = 100;
    // 이렇게 초기화하면 안된다.

    printf("%p\n", ++int_ptr);
    printf("%p\n", ++int_ptr);
    printf("%p\n", ++int_ptr);
}
```

Pointer

```
main() {
    double * double_ptr = 100;
    // 이렇게 초기화하면 안된다.

    printf("%p\n", ++double_ptr);
    printf("%p\n", ++double_ptr);
    printf("%p\n", ++double_ptr);
}
```

인자의 전달

Argument

- 함수를 호출할 때 전달하는 값
- 지역변수이다.

변수의 값을 바꾸고 싶다

```
int add1000(int a) {
    a += 1000;
}

main() {
    int a = 1000;

    printf(“%d”, a);
    add1000(a);
    printf(“%d”, a);
}
```

Call by Value

- 함수의 인자로 값을 전달한다.

Call by Value

변수가 아니라 변수에 들어있는 값이 전달되기 때문에,
절대 변수에 들어있는 값이 바뀌지 않는다.

Call by Value

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

main() {
    printf(“%d”, add(7, 2));
}
```

Call by Reference

- 함수의 인자로 변수의 주소값을 전달한다.
- 변수에 직접 접근할 수 있도록 한다.

Call by Reference

```
int add(int * a, int * b, int * c) {
    * c = * a + * b;
}

main() {
    int a = 26, b = 38;
    int c;

    add(&a, &b, &c);

    printf("%d", &c);
}
```

다음 수업 준비

복습 및 과제

- 오늘 수업 내용 복습
- 과제 반드시 제출
  - 질문은 얼마든지 가능하니 반드시 제출
- 학습시 리눅스 이용

다음 수업 예습

- 포인터의 모든 것
  - 포인터와 배열
  - 함수 포인터
  - …

C Programming – Day 2

JunGu Kang
Dept. of Cyber Security

아주대학교 Whois