

C Programming - Day 1

JunGu Kang
Dept. of Cyber Security



아주대학교



오리엔테이션



아주대학교



학습 목표

- 리눅스에 익숙해지기
 - GUI 없는 세상에 적응
 - Vim 에디터로 작업하고,
 - GCC로 코드를 컴파일
- C 코드를 쉽게 읽고 쓰기
 - 어떤 코드가 좋은 코드인지 알고, 좋은 코드를 작성하기

계획

- 1일차(8월 28일)
 - `printf`, `scanf`, 연산자, 변수/상수 선언, 자료형, 분기문, 반복문
- 2일차(8월 31일 → 8월 30일)
 - 함수, 배열
- 3일차(9월 4일)
 - 포인터 1

계획

- 4일차(9월 7일)
 - 포인터 2
- 5일차(9월 11일)
 - 구조체, 공용체, 사용자 정의 자료형, 열거형
- 6일차(9월 14일)
 - 파일 입출력, 문자열 다루기, 헤더파일

과제

- 매 수업마다 과제 출제
 - github 저장소에 발표자료와 함께 업로드
- 정해진 파일명으로 저장
 - Style(K&R 또는 BSD, Soft Tab)
- 간략한 보고서 작성
 - 보고서 양식 github 저장소에서 다운로드
 - 반드시 pdf로 제출

과제

- 코드와 보고서 모두 압축해서 슬랙 #submit 채널에 업로드
 - 코드 복사해서 붙여넣지 말고,
 - 코드 파일 모두 압축해서 한 번에 업로드
- 과제도 반드시 리눅스에서 Vim으로 작업

질문 방법

- 질문은 슬랙 #qna 게시판에 작성
 - 반드시 코드 첨부
 - 절대 사진 찍어서 업로드하지 말고,
 - 코드 파일 그대로 업로드
 - 오류가 발생할 경우 오류 내용 첨부
- 코멘트에 질문 내용 작성
- 답변은 누구나 작성 가능
 - 답변 작성 시 댓글로 작성

코딩 스타일

- GNU Style
- K&R Style
- BSD Style

코딩 스타일 - GNU 스타일

```
main()
{
    if(1 == 1)
    {
        printf("1")
    }
}
```

코딩 스타일 - K&R 스타일

```
main() {  
    if(1 == 1) {  
        printf("1");  
    }  
}
```

코딩 스타일 - BSD 스타일

```
main()
{
    if(1 == 1)
    {
        printf("1");
    }
}
```

코딩 스타일 - 탭

- Soft Tab : 탭을 스페이스 2개 또는 4개로 사용
- Hard Tab : 탭을 탭 문자로 사용
 - 절대 하지 말 것
 - 서로 다른 운영체제 / 에디터에서 여는 경우 문제 발생

코딩 스타일

- 과제 / 질문 코드 작성시
 - K&R Style 또는 BSD Style 중 선택
 - Soft Tab으로 하되 스페이스 4개로 사용

Vim 에디터



아주대학교



모드

- 편집 모드
- 명령 모드

단축키

version 1.1
April 1st, 06

vi / vim 단축키 모음

Esc
명령 모드

~ 대소문자 전환	! 외부 명령	@ 매크로 실행	# 이전 검색	\$ 줄 끝으로 이동	% 일치하는 괄호 찾기	^ 줄의 첫 글자	& :s 반복	* 다음 검색	(문장 시작) 문장 끝	_ 아래줄로 이동	+ 다음 줄
· 매크로 이동	1	2	3	4	5	6	7	8	9	0 줄의 처음	- 이전 줄	= 자동 들여쓰기
Q 실행 모드	W 다음 WORD	E 끝 WORD	R 수정 모드	T 뒤로 검색	Y 줄단위 복사	U 줄단위 실행취소	I 줄 시작에서 삽입	O 행 위에 삽입	P 커서 이전에 붙여넣기	{ 문단 시작	}	문단 끝
q 매크로 기록	w 다음 단어	e 단어 끝	r 한 문자 교체	t 한 문자 검색	y 복사	u 실행취소	i 편집 모드	o 행 아래에 삽입	p 커서 이후에 붙여넣기	[기타]	기타
A 줄 끝에 덧붙이기	S 줄 삭제후 편집모드	D 줄 끝까지 삭제	F 뒤로 검색	G 파일끝으로 이동	H 화면 상단	J 줄 합치기	K 다음알	L 화면 하단	: ex 명령줄	" 레지스터 지정	열 이동	
a 덧붙이기	s 단어 삭제후 편집모드	d 1,3 삭제	f 한 문자 찾기	g 확장 명령	h ←	j ↓	k ↑	l →	; t/T/f/F 명령 반복	' 매크로 이동	\ 사용 안함	
Z 종료	X 백스페이스	C 줄 끝까지 바꾸기	V 줄단위 비주얼모드	B 이전 WORD	N 이전 (찾기)	M 화면 가운데	< 3 내어쓰기	> 3 들여쓰기	? 찾기 (뒤로)			
Z 확장 명령	X 글자 삭제	c 1,3 바꾸기	v 비주얼 모드	b 이전 단어	n 다음 (찾기)	m· 마크 설정	, 역순 검색	. 명령 반복	/ 찾기			

동작 커서를 이동하거나, 연산자가 동작할 범위를 지정합니다.

명령 바로 동작하는 명령, 빨간색은 편집 모드로 변경됩니다.

연산자 이동 관련 문자(숫자나 커서 이동)와 함께 사용해야 하며, 커서의 위치부터 목적지까지 연산합니다.

확장 특별한 키 함수로, 추가적인 키 입력이 필요합니다.

q· 입력후 (숫자를 제외한)으로 끝낼수 있는) 글자를 입력하여야 합니다.

words: 구분자로 공백, 특수기호 모두 사용

WORDS: 구분자로 공백 문자만 사용

words: quux(foo, bar, baz);

WORDS: quux(foo, bar, baz);

주요 명령행 명령 ('ex'):

:w (저장), :q (종료), :q! (저장하지 않고 종료)

:e f (파일 f 열기),

:%s/x/y/g (파일 전체에서 'x' 를 'y' 로 교체),

:h (vim 도움말), :new (새 파일)

그외 중요한 명령들:

CTRL-R: 재실행 (vim),

CTRL-F/-B: 페이지 위로/아래로,

CTRL-E/-Y: 줄 스크롤 위로/아래로,

CTRL-V: 블록-비주얼 모드 (vim 전용)

비주얼 모드:

커서를 움직여 지정한 범위에 연산자를 적용합니다. (vim 전용)

참고:

(1) 복사/붙여넣기/지우기 명령어를 사용하기 전에 "x"를 입력하여 레지스터(클립보드)를 지정하세요. (x는 a에서 z 또는 * 을 사용할 수 있음) (예: "ay\$ 를 입력하면 현재 커서에서 라인 끝까지의 내용을 레지스터 'a'에 저장합니다.)

(2) 어떤 명령을 입력하기 전에 횟수를 지정하면, 횟수만큼 반복하게 됩니다.(예: 2p, d2w, 5i, d4j)

(3) 연속으로 입력하는 명령은 현재의 라인에 반영됩니다. 예시: dd(현재 라인 지우기), >>(들여쓰기)

(4) ZZ 는 저장후 종료, ZQ는 저장하지 않고 종료.

(5) zt : 커서가 위치한 곳을 제일위로 올리기, zb : 바닥으로, zz : 가운데로

(6) gg : 파일의 처음으로(Vim 전용), gf : 커서가 위치한 곳의 파일 열기(Vim 전용)

vi/vim 에 대한 더 많은 강좌나 팁을 얻으려면 www.viemu.com (ViEmu, MS 비주얼 스튜디오를 위한 vi/vim 에뮬레이션)을 방문하십시오.

단축키

- Esc : 명령 모드로 전환
- a : 덧붙이기(편집 모드로 전환)
- i : 삽입(편집 모드로 전환)
- d : 삭제
- dd : 현재 줄 삭제

명령어

- :w : 저장
- :q : 종료
- 두 명령어를 합칠 수 있음
 - :wq : 저장 후 종료
- 명령에 !를 붙이면 오류가 발생해도 강제로 실행

명령어

- `:set nu` : 줄 번호 보이게 설정
- `:set et` : 탭을 스페이스로 변경

Vim 설정

- `/.vimrc` 파일에서 설정
- Soft Tab 설정
 - `set ts=4`
 - `au Bufenter *.\(c\|cpp\|h\) set et`
 - .c, .cpp, .h 파일인 경우 적용
 - `set et`
 - 모든 파일에 적용
- 줄 번호 보이게 설정
 - `set nu`

C 언어 소개



아주대학교



역사

- 1972년 Dennis M. Ritchie가 Unix 운영체제를 만들기 위해 개발
- B언어 다음에 만들어져서 C언어
- 1979년 K&R C
 - Brian W. Kernighan과 Dennis M. Ritchie가 쓴 The C Programming Language에 쓰임
 - 공식적인 표준은 아님

역사

- 1989년 C89
 - ANSI(American National Standards Institute)에서 표준 제정
 - 1990년 ISO에서 이를 승인(약간의 변경, C90)
 - C89와 C90은 본질적으로 같음
- 1995년 C95
 - ISO가 제정, ANSI가 승인
- 2000년 C99
 - ISO가 제정, ANSI가 승인
- 현재 최신 표준은 C11

특징

- 쉽고 간결한 문법
- 배우기 쉬움
- 이식성이 좋음
 - 특정 아키텍처에 종속적이지 않음(Machine Independent)
- 프로그래머를 믿음
 - 프로그래머가 모든 것을 제어하며, 컴파일러가 자동으로 처리해주지 않음
- 포인터를 이용한 메모리 제어

컴퓨터의 데이터 처리



아주대학교



2진수

0	1	0	1	1	1	1	0
---	---	---	---	---	---	---	---

$$0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 94$$

정수 데이터

59	0	0	1	1	1	0	1	1
198	1	1	0	0	0	1	1	0

정수의 부호

1의 보수?

59	0	0	1	1	1	0	1	1
-59	1	1	0	0	0	1	0	0

정수의 부호

59

0	0	1	1	1	0	1	1
---	---	---	---	---	---	---	---

+

-59

1	1	0	0	0	1	0	0
---	---	---	---	---	---	---	---

=

???

1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

정수의 부호

덧셈의 역원을 더했는데 0이 아니다.

정수의 부호 - 2의 보수

59

0	0	1	1	1	0	1	1
---	---	---	---	---	---	---	---

+

-59

1	1	0	0	0	1	0	1
---	---	---	---	---	---	---	---

=

0

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

문자의 처리 - ASCII

A	65	0	1	0	0	0	0	0	1
---	----	---	---	---	---	---	---	---	---

실수 데이터

0	0	1	1	1	1	0	1	0	1	1	1	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

정수
61

실수
114

61.114

실수 데이터

이게 과연 좋은 방법일까?

실수 데이터

서로 다른 실수끼리 어떻게 연산을 해야 하지?
모든 실수를 같은 크기의 공간으로 나타낼 수 있을까?

Floating Point

$$a \times 2^b$$

실수 데이터

Floating Point

0	0	1	1	1	0	1	1	1	0	0	0	1	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

부호

지수

가수

부동소수점 오차

부동소수점 표현 방식으로는
실수를 정확하게 표현할 수 없다.

부동소수점 오차

이 식을 만족하는 정수 a 와 b 는?

$$0.1 = a \times 2^b$$

Endian

0x12345678

Big Endian

12	34	56	78
----	----	----	----

Little Endian

78	56	34	12
----	----	----	----

변수와 상수



아주대학교



변수와 상수

- 데이터를 저장하기 위한 메모리 공간
- 변수
 - 저장된 값의 수정이 가능
- 상수
 - 저장된 값이 수정이 불가능
 - 선언과 동시에 초기화 해야 함

변수와 상수의 이름

- 알파벳과 숫자 사용 가능
 - 첫 글자는 문자여야 함
 - Underscore(“_”)는 문자로 취급함
 - 그러나 Underscore로 시작해서는 안됨
- 대소문자는 구분함
 - 변수는 소문자, 상수는 대문자
- C는 일반적으로 Snake Case를 사용함
- 예약어는 사용할 수 없음

데이터를 어떻게 읽고 쓸 것인가?

자료형

1	1	1	0	0	0	0	0	0	1	0	0	0	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

2Byte씩 읽으면 {57415}

1Byte씩 읽으면 {224, 71}

자료형

1	1	1	0	0	0	0	0	0	1	0	0	0	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

2Byte씩 정수로 읽으면 {57415}

2Byte씩 실수로 읽으면 ...

자료형

- 정수형
 - char, short, int, long
- 실수형
 - float, double, long double

자료형 - 크기

- 크기가 표준으로 정해져 있는 것은 아님
- 정해진 기준은 다음과 같음
 - short와 int는 최소 2Byte, long은 최소 4Byte
 - short는 int 이하, long은 int 이상
 - float는 single precision
 - double은 double precision

자료형 - 일반적인 크기

Type	Size(x84)	Size(x64)
char	1Byte	1Byte
short	2Byte	2Byte
int	4Byte	4Byte
long	4Byte	8Byte
float	4Byte	4Byte
double	8Byte	8Byte
long double	16(12)Byte	16Byte

변수의 선언

```
main() {  
    char a;  
    short b;  // short int b;  
    int c;  
    long d;  // long int b  
    float f;  
    double g;  
    long double h;  
}
```

변수의 선언

```
main() {  
    char a, b, c;  
    short d, e, f;  
    int g, h, i;  
    long j, k, l;  
    float m, n, o;  
    double p, q, r;  
    long double s, t, u;  
}
```

변수의 선언과 초기화

```
main() {  
    int number1 = 10;  
    double number2 = 3.72;  
}
```

변수의 선언과 초기화

```
main() {  
    int number1;  
    double number2;  
    number1 = 10;  
    number2 = 3.72;  
}
```

변수의 선언과 초기화

- 지역변수는 초기화하지 않으면 알 수 없는 값이 들어있다.
- 전역변수는 초기화하지 않으면 0으로 초기화된다.
 - static 변수도 마찬가지

상수

- 리터럴 상수(Literal Constant)
 - 코드 그대로의 상수
 - 문자 상수(Character Constant)
 - 문자열 상수(String Constant, String Literal)
- 심볼릭 상수(Symbolic Constant)
 - 변수처럼 선언되어 이름을 가지는 상수
- 매크로 상수(Macro Constant)
 - 전처리기 매크로로 선언된 상수
- 열거형(Enumeration)

리터럴 상수

```
main() {  
    int a = 83;    // 83은 리터럴 상수  
    printf("%d", a + 68);    // 68도 리터럴 상수  
}
```

리터럴 상수에도 자료형이 있다

```
main() {  
    int a = 38;    // int(default)  
    long b = 83L;  // long  
    unsigned long c = 26UL; // unsigned long  
    float d = 72F; // float  
    double e = 29.0 // double(default)  
    long double f = 93.1L; // long double  
    double g = 2.3e-1; // double 0.23  
}
```

8진수와 16진수

```
main() {  
    int octal = 026;  // 8진수 26, 10진수로는 22  
    int hexadecimal = 0x37;  // 16진수 37, 10진수로는 55  
}
```

문자 상수

```
main() {  
    int A = 'A';    // 65  
    int a = 'a';    // 97  
}
```

문자 상수

```
main() {  
    int A_char = 'A';    // 'A' == 65  
    int A_octal = '\081' // 0x81 == 65  
    int A_hexadecimal = '\x41' // 0x41 == 65  
    // 모두 같은 표현  
}
```

문자열 상수

- 0개 이상의 문자들의 Sequence
- 맨 뒤에 Null 문자가 붙음
 - 문자열의 끝이 어디인지 나타내기 위함

문자열 상수

```
main() {  
    char str1[] = "hello, world"; // string constant  
    char str2[] = "hello," " world"; // 컴파일시에 이어붙여져 위와 같다  
}
```

문자 상수와 문자열 상수

‘a’와 “a”는 서로 다르다.

심볼릭 상수

심볼릭 상수는 값을 변경할 수 없는 변수
`const`는 변수의 값이 변경되지 않도록 한다.

심볼릭 상수의 선언

```
main() {  
    const char a;  
    const short b;  
    const int c;  
    const long d;  
    const long long e;  
    const float f;  
    const double g;  
    const long double h;  
    // 변수의 선언과 동일하게 하되 const를 붙인다  
}
```

심볼릭 상수의 선언과 초기화

```
main() {  
    const int number;  
    number = 27;  // 상수를 변경하려 했기 때문에 오류 발생  
}
```

심볼릭 상수의 선언과 초기화

```
main() {  
    const int number = 27;  // 올바른 초기화 방법  
}
```

매크로 상수

- 전처리기 매크로로 선언된 상수
- 컴파일 이전에 전처리가 모두 치환한다
 - 결국 치환하고 나면 리터럴 상수...

매크로 상수

```
#define A 65
```

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

Constant Expression

상수만 존재하는 Expression은
모두 컴파일시에 처리된다.

Constant Expression

```
main() {  
    const int a = 999;  
    printf("%d", a + 1);  
}
```


Constant Expression

```
main() {  
    printf("%d", 1000);  
}
```

연산



아주대학교



연산자

- 산술 연산자(+, -, *, /, %)
- 관계 연산자(>, >=, <, <=, ==, !=)
- 논리 연산자(&&, ||)
- 증감 연산자(++, --)
- 비트 연산자(&, |, ^, <<, >>, ~)
- 대입 연산자(+=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=)
- 조건 연산자(?:)

산술 연산자

```
main() {  
    int a = 78, b = 5;  
    printf("%d\n", a + b);  
    printf("%d\n", a - b);  
    printf("%d\n", a * b);  
    printf("%d\n", a / b);  
    printf("%d\n", a % b);  
}
```

관계 연산자

```
main() {  
    int a = 78, b = 5;  
    printf("%d\n", a > b);  
    printf("%d\n", a >= b);  
    printf("%d\n", a < b);  
    printf("%d\n", a <= b);  
    printf("%d\n", a == b);  
    printf("%d\n", a != b);  
}
```

논리 연산자

```
main() {  
    int a = 0, b = 1;  
    printf("%d", a && b);  
    printf("%d", a || b);  
}
```

증감 연산자

```
main() {  
    int a = 5;  
    int b, c, d, e;  
    b = a++;  
    c = ++a;  
    d = a--;  
    e = --a;  
    printf("b: %d, c: %d, d: %d, e: %d", b, c, d, e);  
}
```

증감 연산자는 주의해서 쓰자

한 expression에 증감 연산자는
반드시 하나만 쓰자

증감 연산자는 주의해서 쓰자

Undefined Behavior이기 때문에
컴파일러마다 결과가 제멋대로 나온다.

비트 연산자

```
main() {  
    int a = 5, b = 8;  
    printf("%d", a & b);  
    printf("%d", a | b);  
    printf("%d", a ^ b);  
    printf("%d", a << 1);  
    printf("%d", a >> 1);  
    printf("%d", ~a);  
}
```

대입 연산자

대입 연산자	동작
$a += b$	$a = a + b$
$a -= b$	$a = a - b$
$a *= b$	$a = a * b$
$a /= b$	$a = a / b$
$a \% = b$	$a = a \% b$
$a << = b$	$a = a << b$
$a >> = b$	$a = a >> b$
$a \& = b$	$a = a \& b$
$a = b$	$a = a b$
$a \wedge = b$	$a = a \wedge b$

조건 연산자

- $\text{expression1} ? \text{expression2} : \text{expression3}$
 - expression1 이 참이면 expression2 의 값을 갖고,
 - 거짓이면 expression3 의 값을 갖는다.

조건 연산자

```
main() {  
    int a, b;  
    scanf("%d %d", &a, &b);  
    printf("%d", (a > b) ? a : b);  
}
```

연산의 순서

Operator(연산자)	Associativity(결합 방향)
() [] -> .	왼쪽에서 오른쪽
! ~ ++ - + - * (type) sizeof	오른쪽에서 왼쪽
* % /	왼쪽에서 오른쪽
+ -	왼쪽에서 오른쪽
<< >>	왼쪽에서 오른쪽
< <= > >=	왼쪽에서 오른쪽
== !=	왼쪽에서 오른쪽
&	왼쪽에서 오른쪽
^	왼쪽에서 오른쪽
	왼쪽에서 오른쪽
&&	왼쪽에서 오른쪽
	왼쪽에서 오른쪽
? :	왼쪽에서 오른쪽
= += -= *= /= %= &= ^= = <<= >>=	왼쪽에서 오른쪽
,	왼쪽에서 오른쪽

같은 자료형끼리만 연산할 수 있다.
연산하려면 같은 자료형으로 바꿔야 한다.

형변환

- 암시적 형변환(묵시적 형변환, Implicit)
 - 컴파일러가 알아서 변경
- 명시적 형변환(Explicit)
 - Casting이라고 함
 - 지정한 자료형으로 변경

암시적 형변환

- 작은 자료형을 큰 자료형으로 알아서 바꾼다.
- 같은 자료형끼리는 크기가 큰 쪽으로
 - `int` vs `long` = `long`
 - `float` vs `double` = `double`
- 서로 다른 자료형끼리는 실수형으로
 - `int` vs `float` = `float`
 - 실수를 정수형에 넣으면 소수점 이하가 손실된다.

명시적 형변환

- 캐스팅 연산자 사용
 - (type)
- 내 마음대로 바꿀 수 있다.
 - 실수형을 정수형으로 바꾸는것도 가능

명시적 형변환

```
main() {  
    float a;  
    int b;  
  
    a = 63.34;  
    b = (int) a;  
  
    printf("%d", b);  
}
```

입력받고 출력하기



printf

```
main() {  
    printf("hello, world!");  
}
```

printf

```
main() {  
    int a = 10;  
    printf(a); // 이렇게 쓸 수도 있음  
               // 그러나 FSB 취약점이 존재하기 때문에 사용해서는 안 됨  
}
```

왜 print“f”일까?

Formatting

printf

```
main() {  
    int a = 10;  
    float b = 7.32;  
    printf("a는 %d, b는 %f", a, b);  
}
```


Format String

Format string	Meaning
%d	10진수 정수(Decimal)
%u	부호 없는 10진수 정수(Unsigned Decimal)
%o	부호 없는 8진수 정수(Unsigned Octal)
%x, %X	부호 없는 16진수 정수(Unsigned Hexadecimal)
%c	문자 하나(Single Character)
%s	문자열(String)
%f	10진수 실수(Double)
%e, %E	부동소수점 표현으로 나타낸 10진수 실수(Double)
%g, %G	%f 또는 %g중에서 알아서 결정(Double)
%p	포인터(Pointer)

Escape Sequence

Character	Escape Sequence
비프음(Alert)	\a
백스페이스(Backspace)	\b
폼피드(Formfeed)	\f
줄 바꿈(New Line)	\r
줄 바꿈(Carriage Return)	\n
수평 탭(Horizontal Tab)	\t
수직 탭(Vertical Tab)	\v
백슬래쉬(Backslash)	\\
물음표(Question Mark)	\?
작은따옴표(Single Quote)	\'
큰따옴표(Double Quote)	\"
8진수(Octal Number)	\ooo
16진수(Hexadecimal Number)	\xhh

분기와 반복



아주대학교



Statement와 Block

- Statement : 세미콜론(";")으로 구분되는 Expression
 - $a = b + c;$
 - $a = f(b);$
- Block : 중괄호로 묶인 Statement들의 집합
 - Block은 한 개의 Statement와 같음
 - 중괄호의 끝에 세미콜론을 붙이지 않음

분기 - if

```
main() {  
    if(expression) statement; // 만족하는 경우에만 statement 실행  
}
```

분기 - if

```
main() {  
    if(expression) {  
        statement1;  
        statement2;  
        ...  
    }  
    // block은 한 개의 statement와 동일하므로 가능  
    // 앞으로 따로 언급하지 않아도 가능함  
}
```

조건을 만족하지 않는 경우 - if-else

```
main() {  
    if(expression)  
        statement1;  
    else  
        statement2;  
}
```

조건이 여러개라면?

```
main() {  
    if(expression1)  
        statement1;  
  
    if(expression2)  
        statement2;  
  
    if(expression3)  
        statement3;  
  
    // 과연 좋은 방법일까?  
    // 좋지 않다면 왜?  
}
```


조건이 여러개라면? - if문의 중첩

```
main() {  
    if(expression1) {  
        statement1;  
    } else {  
        if(expression2) {  
            statement2;  
        } else {  
            statement3;  
        }  
    }  
}
```

조건이 여러개라면? - if-else if-else

```
main() {  
    if(expression1) {  
        statement1;  
    } else  
        if(expression2) {  
            statement2;  
        } else {  
            statement3;  
        }  
    // if문 한 개의 statement이므로 중괄호 생략 가능  
}
```

조건의 여러개라면? - if-else if-else

```
main() {  
    if(expression1) {  
        statement1;  
    } else if(expression2) {  
        statement2;  
    } else {  
        statement3;  
    }  
    // 이렇게 붙여 써도 무방하다  
}
```

if-else if-else

if-else if-else문은 결국
if-else문을 중첩한 것에 불과하다.

조건

```
main() {  
    // expression이 참인지 검사하고 싶다면  
  
    if(expression != 0) statement;  
    // 이 표현보다는  
  
    if(expression) statement;  
    // 이 표현이 더 직관적이다.  
}
```

switch

```
main() {  
    switch(variable) {  
        case const-expression1:  
            statement1;  
        case const-expression2:  
            statement2;  
        case const-expression3:  
            statement3;  
        default: // 위 조건을 모두 만족하지 않는 경우 default:로 점프  
            statement4;  
    }  
}
```

switch

```
switch(variable) {  
    case const-expression1:  
        statement1;  
        break; // break가 없으면 아래 statement도 모두 실행됨  
    case const-expression2:  
        statement2;  
        break;  
    case const-expression3:  
        statement3;  
        break;  
    default:  
        statement4;  
        break; // 맨 뒤에는 붙이지 않아도 문제 없지만 붙이는 것이 좋음  
}
```

switch

```
main() {  
    switch(variable) {  
        case const-expression1:  
        case const-expression2:  
            statement1;  
        case const-expression3:  
        case const-expression4:  
            statement2;  
    }  
    // 조건에 따라 점프하는 switch문의 특성상 이런 표현도 가능함  
}
```


switch

```
main() {  
    switch(variable) {  
        case const-expression1:  
            {  
                statement;  
            }  
        // 이렇게 쓰지 않는다.  
    }  
}
```

switch

```
main() {  
    switch(variable) {  
        case another-variable:  
            // 이런 조건을 쓸 수 없다  
            // variable이 상수값과 일치하는지만 확인 가능  
            statement;  
        }  
    }
```

switch

- switch문에서의 조건은 상수 값만 사용할 수 있음
- assembly 수준에서의 구현이 다르기 때문

if vs switch

- switch가 if보다 빠름
- 특히 분기가 많을경우, 조건이 순차적인 경우 빠름
 - if는 매번 cmp하는 반면,
 - switch는 점프테이블을 이용하기 때문

세 가지 반복문(Loop)

- while
- do-while
- for

while

```
main() {  
    while(expression)  
        statement;  
}
```

do-while

```
main() {  
    do  
        statement;  
    while(expression)  
}
```

for

```
main() {  
    for(expression1; expression2; expression3)  
        statement;  
}
```


for

```
main() {  
    int i, j;  
    for(i=0, j=10; i<10; i++, j--)  
        statement;  
}
```

for문을 while문으로

```
main() {  
    expression1;  
    while(expression2) {  
        statement;  
        expression3;  
    }  
}
```

continue와 break

- continue는 중단 후 Loop 맨 앞으로 돌아감
 - 코드 블록을 처음부터 다시 실행
- break는 중단 후 Loop 맨 앞으로 돌아가거나, Switch를 빠져나감
 - 코드 블록을 완전히 빠져나감

continue

```
main() {  
    for(int i = 0; i <= 10; i++) {  
        if(i % 2) continue;  
        printf("%d는 짝수\n");  
    }  
}
```

break

```
main() {  
    int i = 0;  
  
    while(1) {  
        printf("%d\n");  
        if(i == 10) break;  
        i++;  
    }  
}
```

goto

- 코드가 복잡해지니 반드시 필요한 경우가 아니라면 지양
- 절대 쓰지 말아야 하는 것은 아님
 - goto를 쓴 코드가 더 깔끔한 경우 당연히 goto를 써야 함

goto

```
main() {  
    int a;  
    scanf("%d", &a);  
  
    if(a == 1) goto ONE;  
    if(a == 2) goto TWO;  
  
ONE:  
    printf("1");  
TWO:  
    printf("2");  
}
```

다음 수업 준비



아주대학교



복습 및 과제

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

다음 수업 예습

- 함수
- 변수의 Scope
- 배열과 포인터 기초
- 인자의 전달 방식
 - Call by Reference와 Call by Value

C Programming - Day 1

JunGu Kang
Dept. of Cyber Security



아주대학교

