

ROBOTC Programming

using MINDSTORMS EV3

Sangwook Lee
Gyeonggido Office of Education



저작자를 밝히면 이용이 가능하지만, 영리 목적으로 이용할 수 없으며, 내용을 수정해서도 안 됩니다

03 ROBOTC 기초

1. ROBOTC 기본 문법 알아보기
2. 제어 구조와 배열 활용하기

프로그램이란

작성된 **자료구조**에 따라
데이터를 **저장** 하고,

그렇게 저장된

작성된 **알고리즘**에 따라
데이터를 **처리** 하는 것

자료구조란...
데이터를 효율적으로 저장하는 방법
저장 구조 + 입출력 방식

알고리즘이란...
문제를 해결하는 구체적인 절차

프로그래밍이란

```
int main(void)
```

```
{
```

```
// 배열의 선언
```

자료구조

```
int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```
.....
```

```
// 배열에 저장된 값의 합
```

```
for(idx=0; idx<10; idx++)
```

알고리즘

```
    sum += arr[idx];
```

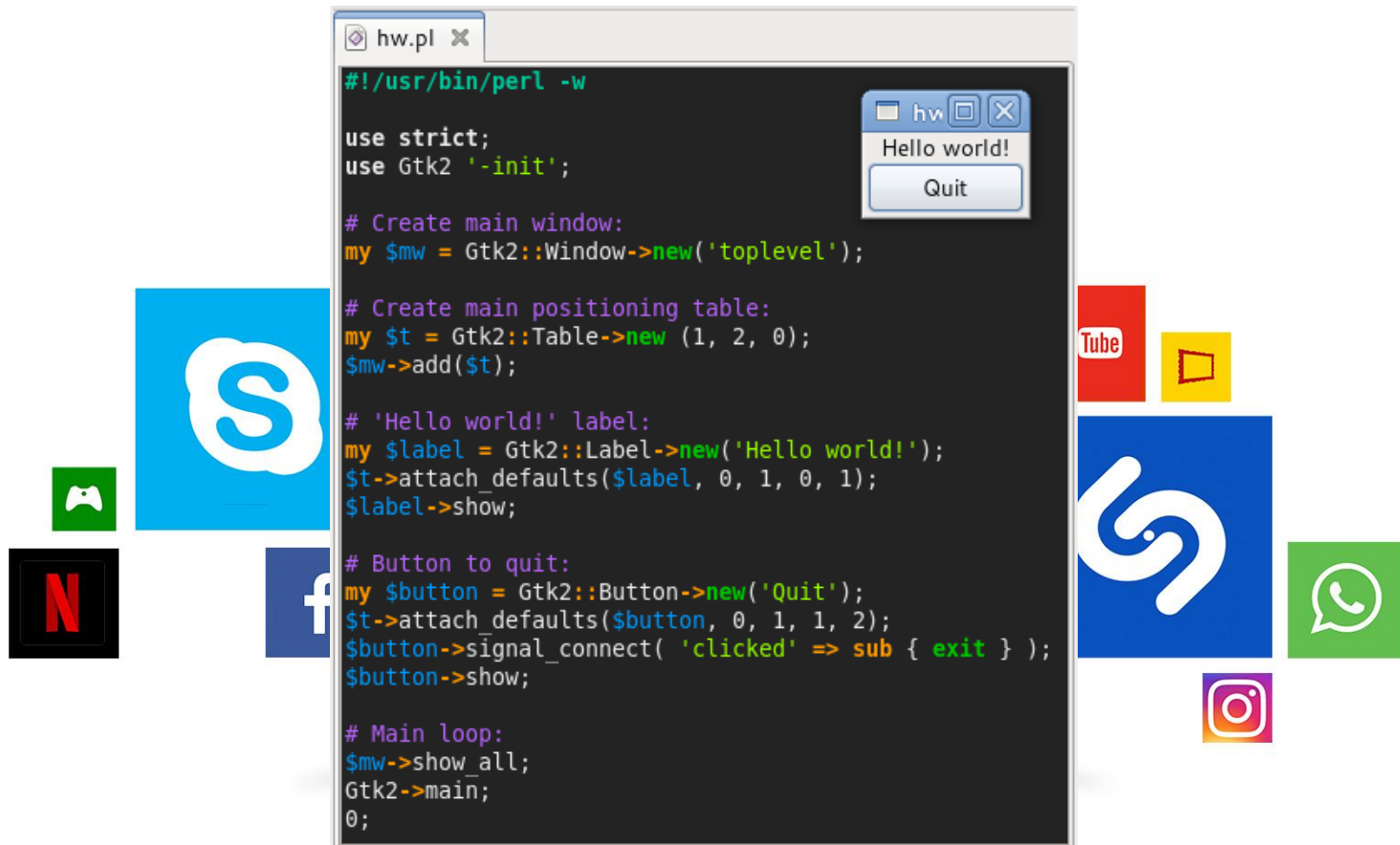
```
.....
```

```
}
```



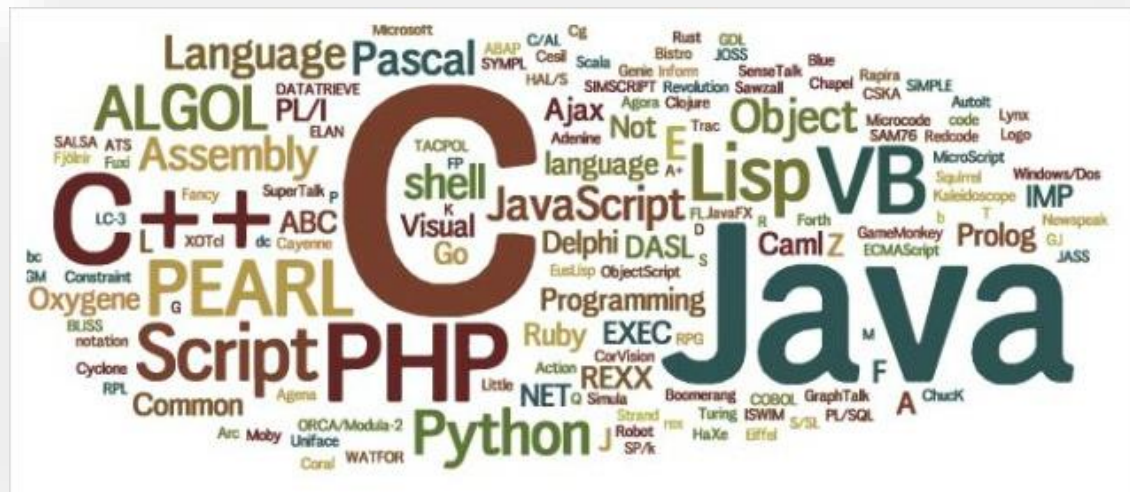
프로그래밍 언어

- 프로그램(program)이란
 - 컴퓨터가 해야 할 일을 지시(작성)한 명령(문장)들의 집합



프로그래밍 언어의 이해

- 프로그래밍 언어 (programming language)
 - 컴퓨터가 해야 할 일을 지시(작성)하기 위해 만든 언어
 - 종류
 - 고수준 언어: C, C++, JAVA, Python, SCRATCH, ...
 - 저수준 언어: 어셈블리, 기계어



프로그래밍 언어의 이해

- 언어 번역 프로그램

- 소스 코드(코딩 결과)를 기계어(실행 파일)로 변환하는 프로그램



- 종류

- 저수준 언어 번역 프로그램: 어셈블러
- 고수준 언어 번역 프로그램: C 컴파일러, JAVA 컴파일러 등

프로그래밍 언어의 이해

• 저수준 언어 vs 고수준 언어

고수준 언어	저수준 언어
• 사람이 이해하기 쉬운 언어 • 처리(번역) 속도가 느림 • 프로그램 작성이 쉬움 • 호환성 높음 → 다양한 CPU에서 실행	• 기계가 이해하기 쉬운 언어 • 처리(번역) 속도가 빠름 • 프로그램 작성이 어려움 • 호환성 낮음 → 특정 CPU에서만 실행

```
int oddsum(int n) {
    int sum = 0, i = 1;
    while (i <= n) {
        sum = sum + i;
        i = i + 2;
    }
    return sum;
}
```

C Language

```
oddsum: add $a1,$zero,$zero
        addi $v1,$zero,1
        slt $a2,$zero,$a0
        beq $a2,$zero,L7
        add $a1,$a1,$v1
        addi $v1,$v1,2
        slt $v0,$a0,$v1
        beq $v0,$zero,L5
        add $v0,$zero,$a1
        jr  $ra
L5:
L7:
```

Assembly Language

```
00000000 00000000 00101000 00100000
00100000 00000011 00000000 00000001
00000000 00000100 00110000 00101010
00010000 11000000 00000000 00000101
00000000 10100011 00101000 00100000
00100000 01100011 00000000 00000010
00000000 10000011 00010000 00101010
00010000 01000000 11111111 11111100
00000000 00000101 00010000 00100000
00000011 11100000 00000000 00001000
```

Machine Language

고수준 언어가 기계어로 번역되는 과정

C compiler
for PowerPC

```
int main()
{
    int a, b, c;
    a = 3;
    b = 9;
    c = a + b;
    return c;
}
```

C language

C compiler
for Core i7

```
li    v0,3
sw    v0,8(s8)
li    v0,9
sw    v0,4(s8)
lw    v1,8(s8)
lw    v0,4(s8)
nop
addu  v0,v1,v0
sw    v0,0(s8)
```

assembly language

assembler



PowerBook G4
(CPU: PowerPC)

```
24 02 00 03
af c2 00 08
24 02 00 09
af c2 00 04
8f c3 00 08
8f c2 00 04
00 00 00 00
00 62 10 21
af c2 00 00
```

machine code

```
movl  $0x3,-0x10(%ebp)
movl  $0x9,-0xc(%ebp)
mov   -0xc(%ebp),%edx
mov   -0x10(%ebp),%eax
add   %edx,%eax
mov   %eax,-0x8(%ebp)
```

assembly language

assembler

```
c7 45 f0 03 00 00 00
c7 45 f4 09 00 00 00
8b 55 f4
8b 45 f0
01 d0
89 45 f8
```

machine code

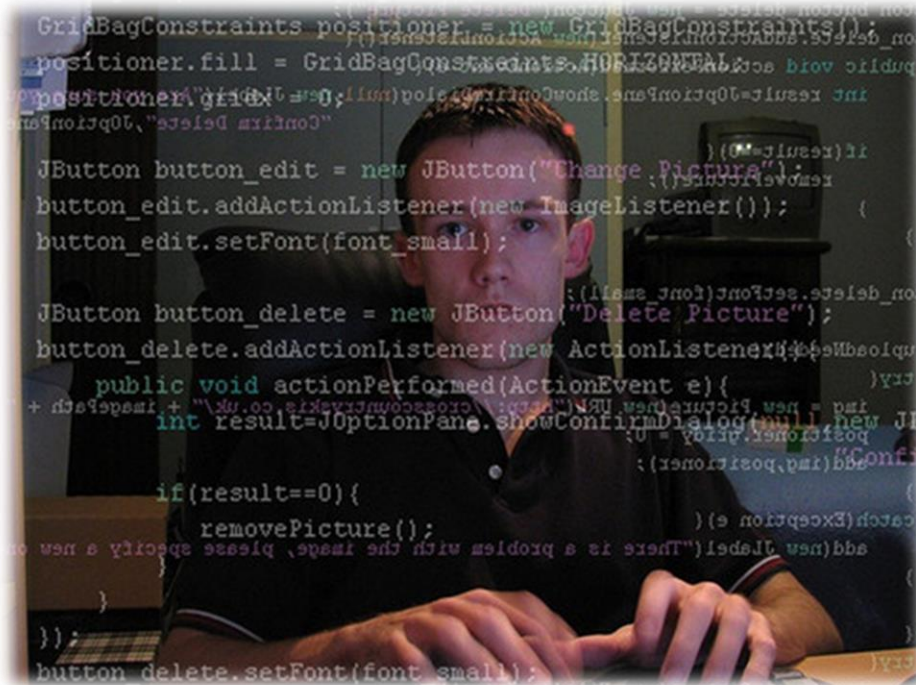


LG UltraGear
(CPU: Core i7)

프로그래밍 언어의 이해

★ 코딩(coding)이란

- 어떤 문제를 해결하기 위해, 프로그래밍 언어를 사용하여 프로그램을 작성하는 것



프로그래밍 언어의 이해



“모든 사람은 코딩을 배워야 합니다.
컴퓨터 프로그램은 생각하는 방법을
가르쳐 주기 때문입니다.”

_ 스티브 잡스(애플 창업자)

“다음 세대는 프로그래밍을 읽기와
쓰기처럼 가르치고 있을 겁니다.”

_ 마크 주커버그(페이스북 창업자)



프로그래밍 언어의 이해

ROBOTC 규칙을 이용하여 작성된 명령(문장)들도
사람만이 이해할 수 있다.

이러한 명령(문장)들을 EV3에서 실행되려면

ROBOTC 컴파일러에 의해

EV3가 이해할 수 있는 기계어로 변환되어야 한다.

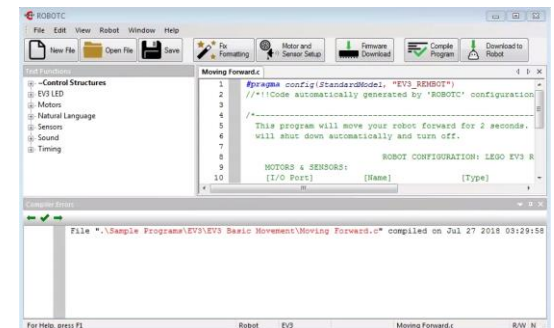
ROBOTC 개요

- C란

- 1972년 벨 연구소의 데니스 리치가 개발
- JAVA와 함께 가장 많이 사용되는 프로그래밍 언어

- ROBOTC란

- 카네기 멜론 대학에서 개발한 로봇 제어용 프로그래밍 언어
- C 언어 규칙을 따르는 텍스트 기반의 프로그래밍 언어
- 프로그램 개발과 디버깅이 가능한 IDE(통합개발환경) 제공



ROBOTC 프로그램 작성 규칙

1. 적어도 하나의 main() 함수가 필요

👉 ROBOTC에서는 **task**형 main 함수 안에 문장을 기술

```
#pragma config(StandardModel, "EV3_REMBOT")  
//EV3_REMBOT 표준 모델로 모터와 센서 값 설정  
  
task main()  
{  
    setMotorSpeed(leftMotor, 50); //왼쪽 모터 출력 50%  
    setMotorSpeed(rightMotor, 50); //오른쪽 모터 출력 50%  
    sleep(2000); //현 상태를 2000ms(2초) 동안 유지  
}
```

<2초 동안 전진하는 프로그램>

ROBOTC 프로그램 작성 규칙

2. 전처리 구문은 main 함수 밖에 따로 선언



```
#pragma config(StandardModel, "EV3_REMBOT")
```

```
task main()  
{  
    setMotorSpeed(leftMotor, 50);  
    setMotorSpeed(rightMotor, 50);  
    sleep(2000);  
}
```

<2초 동안 전진하는 프로그램>

ROBOTC 프로그램 작성 규칙

※ 전처리 구문

- main() 함수보다 먼저 처리되는 구문
- 종류
 - include: 다른 파일 내용 포함
예) `#include <stdio.h>`
 - define: 상수 정의
예) `#define SIZE 1024`
 - pragma: 컴파일러에게 지시
예) `#pragma config(...);`

ROBOTC 프로그램 작성 규칙

전처리 구문에서 상수를 정의할 경우 좋은 점

1. 값을 숫자가 아니라 단어로 표현하기 때문에
프로그램 속에 있는 값의 의미를 이해하기 쉬움
2. 전처리 구문에서 값을 일괄적으로 변경할 수 있기 때문에
프로그램 속에 있는 값의 변경이 쉬움

ROBOTC 프로그램 작성 규칙

3. 대문자와 소문자를 엄격하게 구분

- ROBOTC가 제공하는 예약어나 함수를 사용할 때 대·소문자를 구분하지 않으면 오류 발생

※ 예약어(reserved word) = 키워드(keyword)

- ROBOTC에서 사용하기 위해 미리 만들어 놓은 단어
- 종류
 - 자료 정의 명령 관련: int, float, ...
 - 제어 명령 관련: if, switch, for, while, ...
 - 전처리 명령 관련: define, include, ...

ROBOTC 프로그램 작성 규칙

4. 모든 문장은 세미콜론(;)으로 종료

5. 공백과 줄바꿈은 프로그램 실행에 영향을 미치지 않음

그러나...

프로그램의 가독성을 높이기 위해서는

적절한 공백과 줄바꿈이 필요!

(퀴즈)

- 아래 프로그램은 실행이 될까?

```
taskmain(){displayBigTextLine(1,"Hello,EV3!");sleep(5000);}
```

안 됨

시간 지연 함수 sleep()

- sleep 함수는

- 특정 시간 동안 다음 코드로 넘어가지 않고 지연시킴
- 괄호 안에 지연 시간을 입력
- 지연 시간 단위는 1ms(0.001초)
- 주로 모터 값을 유지시킬 때 사용
- 동일한 역할을 하는 함수로 wait1Msec() 함수가 있음

[예제] 2초간 전진

1. 새 파일 생성

[File]-[New]-[New File]

2. 코드 작성

```
task main()  
{  
    setMotorSpeed(motorB, 50);  
    setMotorSpeed(motorC, 50);  
    sleep(2000);  
}
```

[예제] 2초간 전진

3. 저장

[File]-[Save]

4. 컴파일 & 다운로드

[Robot]-[Compile and Download Program]

☞ 단축키: <F5>

5. 실행

Program Debug 창에서 [Start] 클릭

(퀴즈)

- 아래 프로그램의 실행 결과는?

```
task main()  
{  
    setMotorSpeed(motorB, 50);  
    setMotorSpeed(motorC, 50);  
}
```

움직이지 않음

데이터 저장과 출력

- 변수란

- 프로그램이 실행 되는 동안 값을 저장하기 위해 사용하는 메모리 공간

- 변수명 규칙

- 대문자(A~Z), 소문자(a~z), 숫자(0~9), 밑줄(_)을 사용
- 숫자로 시작할 수 없음
- 대문자와 소문자 구분

예) 'robot'과 'ROBOT'은 다른 변수

- ~~예약어와 함수명은 사용 불가~~

☞ 일부는 맞고 일부는 틀린 말... C언어에서 전처리 관련 예약어와 함수명은 변수명으로 사용하여도 오류가 나지 않음

데이터 저장과 출력

- TIP

C언어에서

자료 정의 관련 예약어(int, float, ...)와
제어 관련 예약어(if, switch, for, ...)는
변수로 사용할 수 없지만

전처리 관련 예약어(include, define, ...)는
변수로 사용할 수 있음

그러나, 코드를 이해하는데 어려움이 따를 수 있으므로
사용하지 않는 것이 바람직

데이터 저장과 출력

- TIP

함수명(setMotorSpeed, sleep, ...)은
변수명으로 사용할 수 있다.

하지만...

사용된 변수명은 다시 함수명으로 사용할 수 없다.
따라서, 사용하지 않는 것이 바람직~

데이터 저장과 출력

- 상수란
 - 프로그램이 실행 되는 동안 사용하는 변하지 않는 변수 또는 값
- 상수의 종류
 - 상수(constant)
 - 변하지 않는 변수
 - 예) SIZE, PI, MAX, ...
 - 리터럴(literal)
 - 변하지 않는 값
 - 예) 1024, 3.14, a, Hello, ...

데이터 저장과 출력

- 상수 선언 방법

- #define을 이용한 상수 선언

```
#define 상수명 값
```

ex) #define PI 3.14

- const를 이용한 상수 선언

```
const 자료형 상수명 = 값
```

ex) const float PI = 3.14;

데이터 저장과 출력

- 자료형이란

- 변수에 저장되는 값의 형태

변수 선언 시 자료형 명시 필요!

ex) `int a=92; float x=7.2;`

- 자료형 종류

자료형	의미	선언문	저장 공간 크기
정수형	소수점을 포함하지 않는 수	int	4-byte
실수형	소수점을 포함하는 수	float, double	4-byte, 8-byte
문자형	문자	char	1-byte

[예제] 정수형 변수값 출력

1. 새 파일 생성

[File]-[New]-[New File]

2. 저장

[File]-[Save]

☞ 파일명 입력 후 저장

[예제] 정수형 변수값 출력

3. 코드 작성

```
task main()  
{  
    int x;  
    x = 27;  
    displayBigTextLine(1, "x is %d", x);  
    sleep(5000);  
}
```

[예제] 정수형 변수값 출력

4. 컴파일 & 다운로드

[Robot]-[Compile and Download Program]

☞ 단축키: <F5>

5. 실행

Program Debug 창에서 [Start] 클릭

[예제] 정수형 변수값 출력

• 코드 설명

```
task main()
{
    int x;
    // 정수를 저장하는 변수 x를 만들

    x = 27;
    // x에 27을 저장

    displayBigTextLine(1, "x is %d", x);
    // EV3 화면의 두 번째 줄에 "x is 27"을 출력

    sleep(5000);
    // 출력 화면을 5초 동안 유지
}
```

첫 번째 인자는 출력 줄의 위치 지정

EV3 화면은 작은 글씨(8×8 픽셀)를 기준으로
0~15까지 열여덟 개의 줄이 있음

두 번째 인자는 출력 문장 지정

세 번째 인자는 출력 값 지정
(값은 출력 문장 속의 서식 문자 위치에 출력됨)

출력 문장 속에 있는 서식 문자는…
실제로 출력되지는 않고, 출력할 값의 형식을 나타낸다.

☞ % 의미: 서식 문자 시작

☞ d 의미: 10진수 (o: 8진수, x: 16진수)

[예제] 실수형 변수값 출력

- 코드

```
task main()
{
    float x = 3.141592;

    displayBigTextLine(1, "x is %.2f", x);

    sleep(5000);
}
```

[예제] 실수형 변수값 출력

- 코드 설명

```
task main()  
{  
    float x = 3.141592;  
    // 실수형 변수 x를 선언하고 3.141592로 초기화  
  
    displayBigTextLine(1, "x is %.2f", x);  
    // EV3 화면의 두 번째 줄에 "x is 3.14"를 출력  
  
    sleep(5000);  
}
```

.2의 의미...

변수 x의 값을 소수점 셋째 자리에서 반올림하여
소수점 둘째 자리까지 출력

[예제] 문자형 변수값 출력

- 코드

```
task main()
{
    char x = 'a';

    displayBigTextLine(1, "x is %c", x);

    sleep(5000);
}
```

[예제] 문자형 변수값 출력

- 코드 설명

```
task main()  
{  
    char x = 'a';  
    // 문자형 변수 x를 선언하고 a로 초기화  
  
    displayBigTextLine(1, "x is %c", x);  
    // EV3 화면의 두 번째 줄에 "x is a"를 출력  
  
    sleep(5000);  
}
```

Q) 서식 문자 %c를 %d로 바꾸면 출력 결과는?

[예제] 문자열 변수값 출력

- 코드 설명

```
task main()  
{  
    string x = "ROBOTC language";  
    // 문자열 변수 x를 선언하고 초기화  
  
    displayString(1, "x is %s", x);  
    // EV3 화면의 두 번째 줄에 "x is ROBOTC language"를 출력  
  
    sleep(5000);  
}
```

일반적인 C 언어에서는 문자열 자료형이 없기 때문에
문자열 대신에 문자열이 저장된 주소(포인터)를 저장함

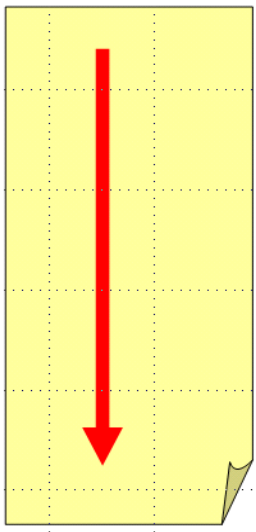
(예)
char *x = "C language";

03 ROBOTC 기초

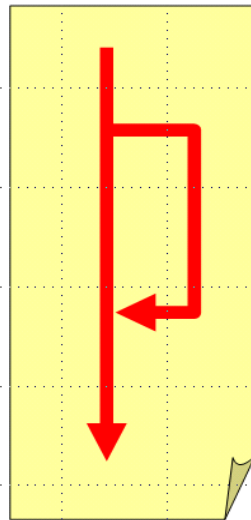
1. ROBOTC 기본 문법 알아보기
2. 제어 구조와 배열 활용하기

제어 구조

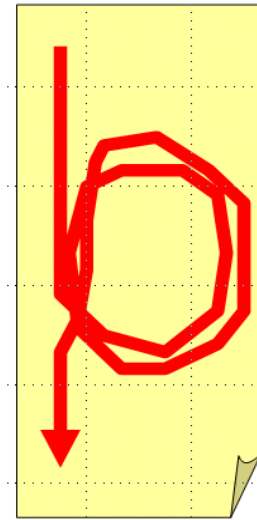
- 제어 구조(control structure)란
 - 프로그램의 처리(실행) 흐름을 제어하기 위해 사용되는 문장의 구조



순차



선택



반복

<프로그램 처리 흐름 유형>

제어 구조

- 제어 구조의 필요성

프로그램을 작성하다 보면

여러 문장 중에서 어떤 문장을 선택해야 되는 경우나,
어떤 문장을 반복적으로 실행하면 좋은 경우가 있는데...



Q) 여러 문장 중 한 문장을 선택해야 되는 경우?

A) 점수에 따라 학점(A, B, C)을 선택하는 경우

Q) 어떤 문장을 반복하여 사용하면 좋은 경우?

A) 구구단 계산 시 한 문장을 반복하여 사용하는 경우

제어 구조

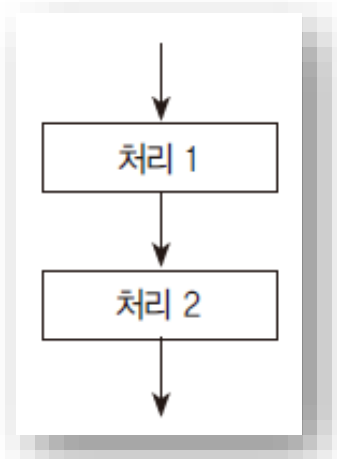
- 제어 구조의 필요성

이때, 선택 또는 반복 구조를 만들면
프로그램의 구조가 간결하게 되어
프로그램을 이해하고 수정하기가 쉬워진다.

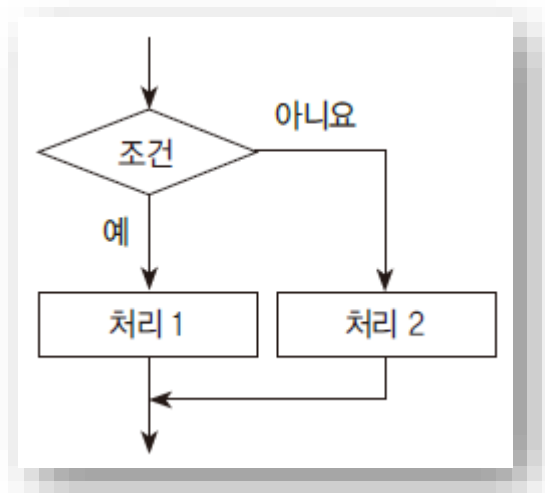
제어 구조

• 제어 구조 종류

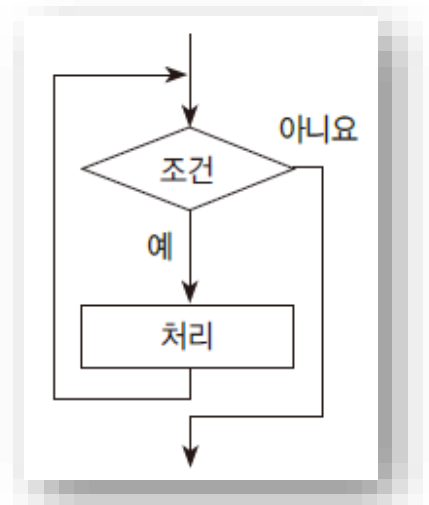
- 순차 구조: 처리 순서가 한 방향으로 진행되는 구조
- 선택 구조: 조건에 따라 처리 순서가 다르게 진행되는 구조
- 반복 구조: 조건을 만족하는 동안 처리가 반복하여 진행되는 구조



순차 구조



선택 구조



반복 구조

제어문

- 제어문이란

- 조건에 따라 특정 문장을 선택하거나 반복하는 제어 구조를 만드는 명령문

- 제어문 종류

- 선택문
 - 선택 구조를 만드는 명령문
예) if, switch
 - 반복문
 - 반복 구조를 만드는 명령문
예) for, while

제어문

- IF문

- 구조

```
if( 조건식 )  
{  
    문장;  
}
```

- 의미

조건식이 참일 때 중괄호 속 문장 실행
(실행 후 다시 조건식의 참·거짓을 판단하지 않음)

제어문

- IF-ELSE문

- 구조

```
if( 조건식 ) {  
    문장A;  
} else {  
    문장B;  
}
```

- 의미

조건식이 참일 때 첫 번째 중괄호 속 문장A 실행하고
조건식이 거짓일 때 두 번째 중괄호 속 문장B 실행

[예제] 터치 센서 누르면 후진

- 코드

```
task main()  
{
```

```
    while(getTouchValue(S1) != 1) {  
        displ  
    }
```

터치센서를 누르지 않았을 경우 무한 대기

```
    if(getTouchValue(S1) == 1) {
```

→ 터치 센서를 눌렀을 때

...

```
        setMotorSpeed(motorB, -50);
```

```
        setMotorSpeed(motorC, -50);
```

} 좌·우 모터의 속도를 -50으로 설정
하여 EV3를 후진

```
        sleep(2000);
```

```
    }
```

```
}
```

좌·우의 값이 같은지 비교하는 비교연산자

제어문

- WHILE문

- 구조

```
while( 조건식 )  
{  
    문장;  
}
```

- 의미

조건식이 참일 때 중괄호 속 문장 실행
(실행 후 다시 조건식의 참·거짓을 판단함)

[예제] 좌·우 회전 3번 반복 후 사운드 출력

• 코드 (while문 사용)

```
task main()
{
    int i = 0;

    while(i<3) {
        setMotorSpeed(motorB, -50);
        setMotorSpeed(motorC, 50);
        sleep(500);

        setMotorSpeed(motorB, 50);
        setMotorSpeed(motorC, -50);
        sleep(500);

        i++;

        setMotorSpeed(motorB, 0);
        setMotorSpeed(motorC, 0);

        playSound(soundBeepBeep);
        sleep(500);
    }
}
```

회전 수 측정 변수 선언 후 0으로 초기화

회전 수가 세 번 미만일 때 중괄호 실행

0.5초간 좌회전

0.5초간 우회전

회전 수 측정 변수값 1 증가

++은 변수값을 1만큼 증가시키는 연산자이고
--은 변수값을 1만큼 감소시키는 연산자이다.

++와 --을 증감연산자라고 한다.

제어문

- FOR문

- 구조

```
for( 초기식; 조건식; 증감식 )  
{  
    문장;  
}
```

- 의미

- ① 초기식 실행
- ② 조건식이 참일 때 중괄호 속 문장 실행
(거짓일 때는 for문을 빠져 나옴)
- ③ 증감식 실행 후, ②번으로 돌아감

[예제] 좌·우 회전 3번 반복 후 사운드 출력

- 코드 (for문 사용)

```
task main()
{
    int i = 0; → ?

    while (i>3) { → ?
        setMotorSpeed(motorB, -50);
        setMotorSpeed(motorC, 50);
        sleep(500);

        setMotorSpeed(motorB, 50);
        setMotorSpeed(motorC, -50);
        sleep(500);

        i++ → ?
    }

    setMotorSpeed(motorB, 0);
    setMotorSpeed(motorC, 0);

    playSound(soundBeepBeep);
    sleep(500);
}
```

배열의 구조

- 배열이란

- 연속적인 기억 장소에 동일한 형식의 자료들을 저장하는 자료 구조
- 여러 자료들을 하나의 배열 이름으로 저장
- 배열 내 각각의 저장 공간은 **첨자**로 구분
- 저장된 자료들에 대하여 반복적인 작업을 할 때 매우 효율적



“배열도 넓은 의미로는 변수!”

배열의 활용

- 배열의 장점

여러 저장 공간을 하나의 이름으로 표현하기 때문에
같은 형식을 가진 여러 자료들의 저장 공간을 한 번에 만들기 쉬움

변수로 만들기	배열로 만들기
<pre>int 김○○; int 이☆☆; int 박♡♡;</pre>	<pre>int 정보[3];</pre>

<변수와 배열을 사용하여 3개의 저장 공간 만들기>

배열의 활용

- 배열의 장점

여러 저장 공간을 첨자로 구분하기 때문에
반복문을 함께 사용하면 **간결한 작업 처리가 가능**

5를 더하기 (순차 구조)	5를 더하기 (반복 구조)
<pre>정보[0] = 정보[0] + 5; 정보[1] = 정보[1] + 5; 정보[2] = 정보[2] + 5; : 정보[9] = 정보[9] + 5;</pre>	<pre>for (x=0; x<=9; x++) { 정보[x] = 정보[x] + 5; }</pre>

<10개의 자료에 5를 더하기>

[예제] 배열 초기화

- 소스 코드

```
task main()  
{  
    int data[5];  
    int i;
```

```
    for(i=0; i<5; i++) // 반복문을 사용하여 배열을 초기화  
    {  
        data[i]=0;  
    }
```

반복문을 사용하여 배열을 0으로 초기화하는 코드

```
    for(i=0; i<5; i++) // 배열의 첨자와 내용을 출력하는 반복문  
    {  
        displayTextLine(1, "data[%d] = %d", i, data[i]);  
        sleep(2000);  
    }  
}
```

Backup Slides

Instructions

- If you want to talk to foreigners, you should be able to speak their languages
 - Likewise, to talk to a computer, you must speak its language
- The words of a computer's language are called **instructions**
- The collection of instructions is called **instruction set**
- **Different CPUs have different instruction sets**
 - x86, MIPS, and ARM have their own instruction sets
 - But, they have many aspects in common

Levels of Program Code

- **High-level language**
 - Level of abstraction closer to problem domain
 - Provides productivity and portability
- **Assembly language**
 - Textual and symbolic representation of instructions
- **Machine code (object code or binary)**
 - Binary bits of instructions and data

High-level
language
program
(in C)

```
swap(int v[], int k)
{int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

Compiler

Assembly
language
program
(for MIPS)

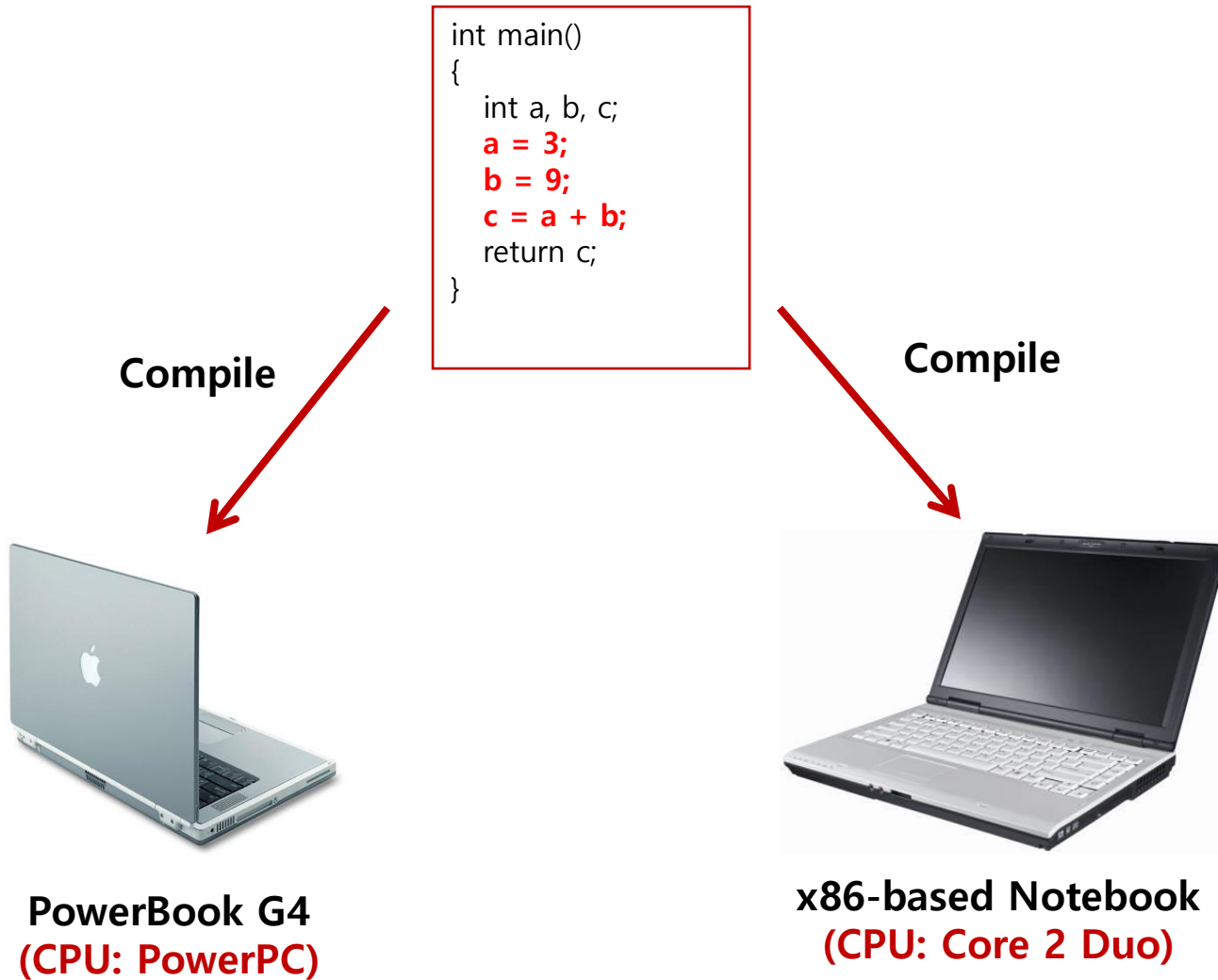
```
swap:
    muli $2, $5, 4
    add  $2, $4, $2
    lw   $15, 0($2)
    lw   $16, 4($2)
    sw   $16, 0($2)
    sw   $15, 4($2)
    jr   $31
```

Assembler

Binary machine
language
program
(for MIPS)

```
000000001010000100000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
100011001111001000000000000000100
101011001111001000000000000000000
101011000110001000000000000000100
00000011111000000000000000001000
```

High-Level Code is Portable



High-level Code to Executable (Binary)

- What steps did you take to run your program (`hello.c`) on your Linux machine?

```
#include <stdio.h>

int main(void)
{
    printf("Hello World!\n");
    return 0;
}
```

```
%gcc -g hello.c -o hello
```

```
    // hello is a machine code (binary or executable)
```

```
    // -g is optional
```

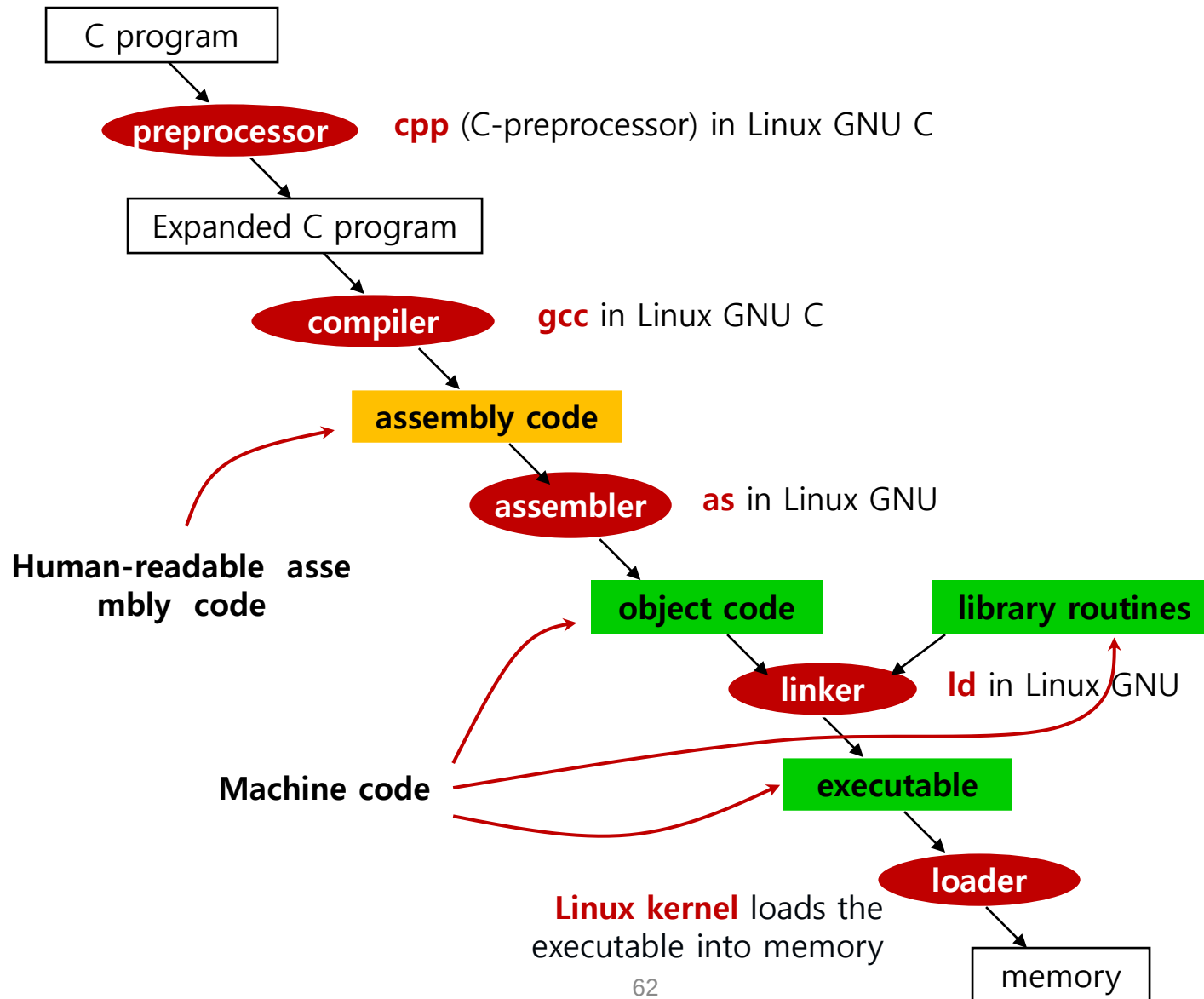
```
%./hello
```

```
% Hello World!
```

```
%objdump -S -D hello
```

```
    // with objdump, you can see human-readable assembly code
```

High-level Code to Executable (Binary)



High-level Code to Executable (Binary)

- The command `gcc` hides all the details
- **Compile `simple_sub.c` with `gcc -v simple_sub.c -o simple_sub`**
 - You will see all the details of what `gcc` does for compilation
- **Compilation goes through several steps to generate a machine code**
 - Preprocessing
 - Compilation
 - Assembler
 - Linker

```
#include <stdio.h>

#define A 3
#define B 5

int main() {
    printf("%d - %d = %d",
           A, B, mysub(A, B))
    ;
    return 0;
}

int mysub(int op1, int op2)
{
    int myres;

    myres = op1 - op2;

    return myres;
}
```

High-level Code to Executable (Binary)

- **Preprocessing**

- Use to expand macros and header files included
- `%cpp simple_sub.c > simple_sub.i`
 - open `simple_sub.i` to see what you got

- **Compilation**

- **Actual compilation of the preprocessed code to assembly language** for a specific processor
- `%gcc -S simple_sub.i`
 - Output will be stored in `simple_sub.s`
 - Open `simple_sub.s` to see what you got

- **Assembler**

- Convert **assembly language** into **machine code** and generate an object file
- `%as simple_sub.s -o simple_sub.o`
 - The resulting file `simple_sub.o` contains the machine instructions for the program, with an undefined reference to `printf`

High-level Code to Executable (Binary)

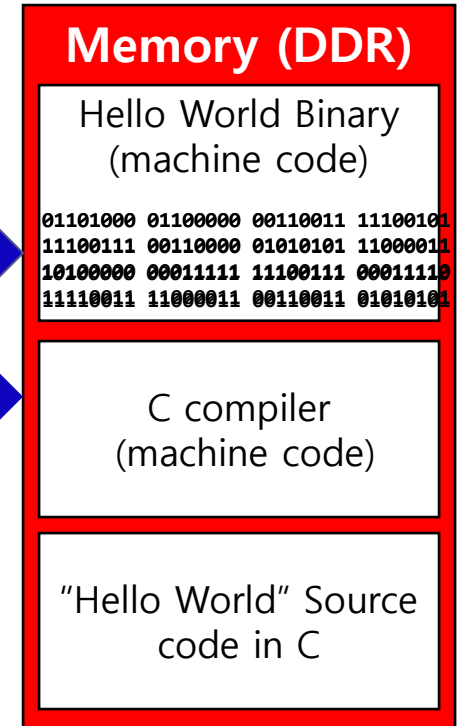
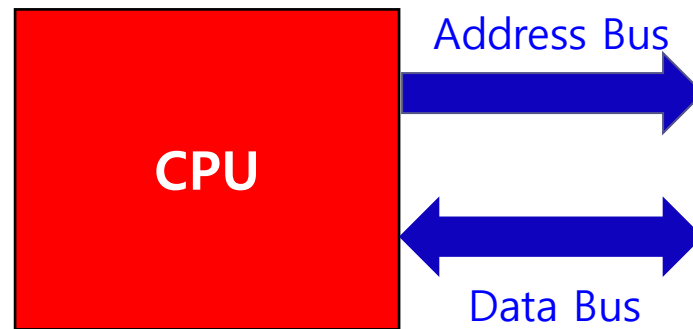
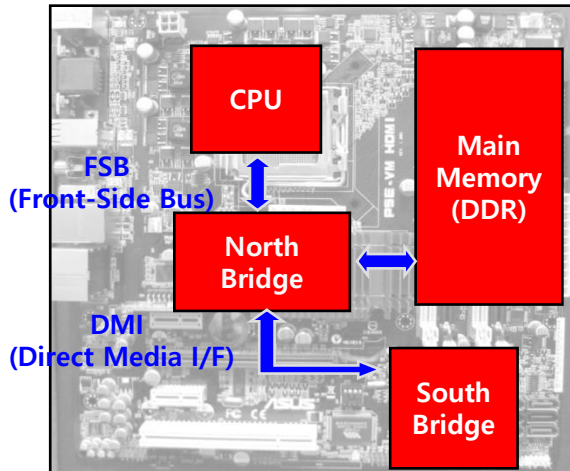
- **Linker**

- Final stage of compilation
- Linking object files to create an executable
- In practice, an executable requires many external functions from system and C run-time (crt) libraries
- Consequently, the actual link commands used internally by GCC are complicated.
- Example
 - `% ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 -z relro /usr/lib/crt1.o /usr/lib/crti.o /usr/lib/gcc/x86_64-linux-gnu/4.4.3/crtbegin.o -L/usr/lib/gcc/x86_64-linux-gnu/4.4.3 -L/usr/lib/gcc/x86_64-linux-gnu/4.4.3 -L/usr/lib -L/lib -L/usr/lib -L/usr/lib/ -L/usr/lib/x86_64-linux-gnu -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed /usr/lib/gcc/x86_64-linux-gnu/4.4.3/crtend.o /usr/lib/crtn.o simple_sub.o -o simple_sub`

- **Now run your program**

- `% ./simple_sub // Linux kernel loads the program into memory`
- `% 3 - 5 = -2 // output`

Stored Program Concept



- Instructions and data are represented in binary
- Instructions and data are stored in memory
- CPU fetches instructions and data to execute
- Binary compatibility allows compiled programs to work on different computers with the same ISA
 - Standardized ISAs

Cross Compiler

- Hmm, sound good so far
- But, wait! We are talking about MIPS (not x86). Then, How to generate the MIPS machine code without a MIPS machine?
- You are still able to generate MIPS binaries on an x86 machine
 - How? Use a **cross-compiler**!!!

Normal compilation

```
int main()
{
    int a, b, c;
    a = 3;
    b = 9;
    c = a + b;
    return c;
}
```



compile

x86 machine code

```
a = 3;
c7 45 f0 03 00 00 00    movl    $0x3,-0x10(%ebp)
b = 9;
c7 45 f4 09 00 00 00    movl    $0x9,-0xc(%ebp)
c = a + b;
8b 55 f4                mov     -0xc(%ebp),%edx
8b 45 f0                mov     -0x10(%ebp),%eax
01 d0                  add     %edx,%eax
89 45 f8                mov     %eax,-0x8(%ebp)
```



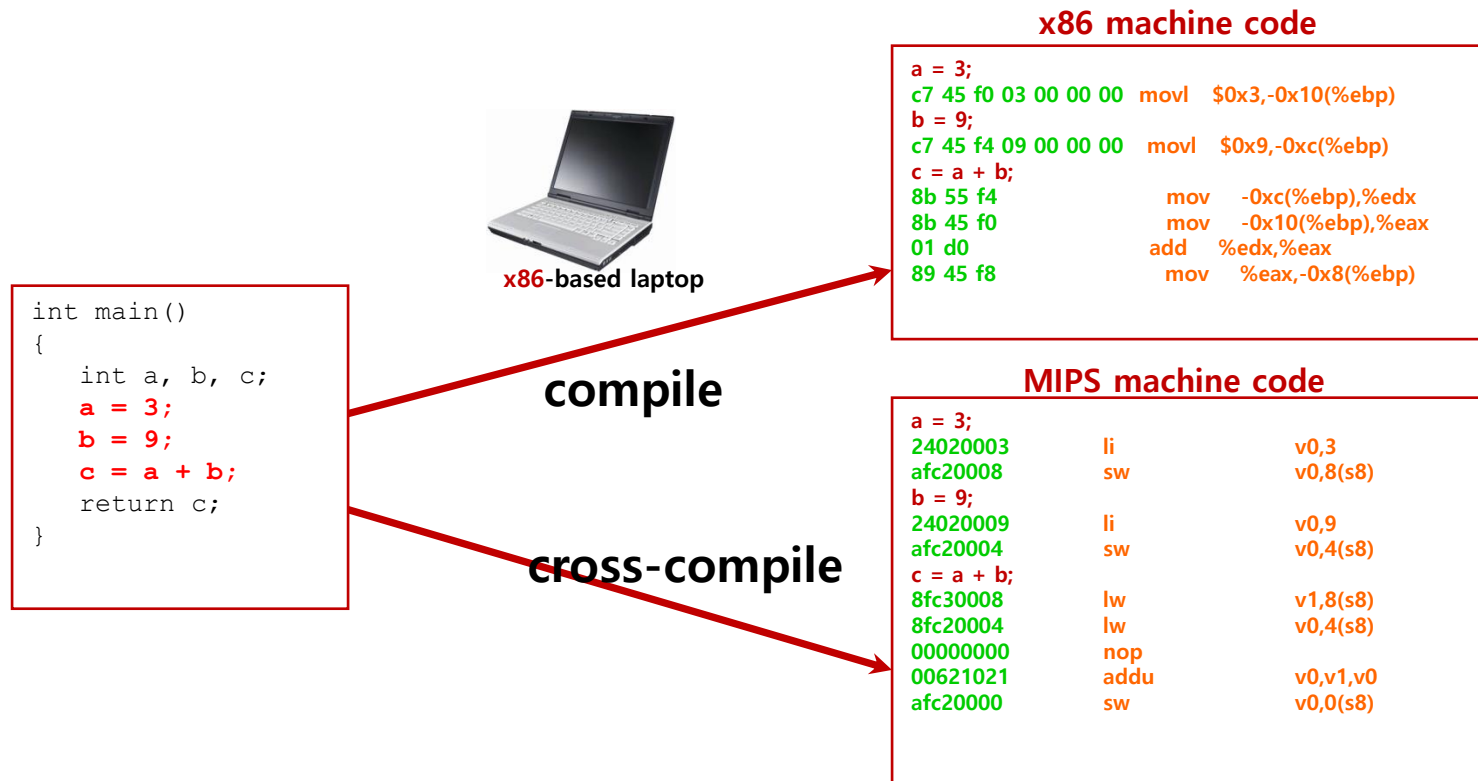
compile

MIPS machine code

```
a = 3;
24020003                li         v0,3
afc20008                sw         v0,8(s8)
b = 9;
24020009                li         v0,9
afc20004                sw         v0,4(s8)
c = a + b;
8fc30008                lw         v1,8(s8)
8fc20004                lw         v0,4(s8)
00000000                nop
00621021                addu        v0,v1,v0
afc20000                sw         v0,0(s8)
```

Cross Compiler (Cont.)

- A **cross compiler** is a compiler capable of creating executable code for a platform other than the one on which the compiler is run -- Wiki



MIPS Cross Compiler

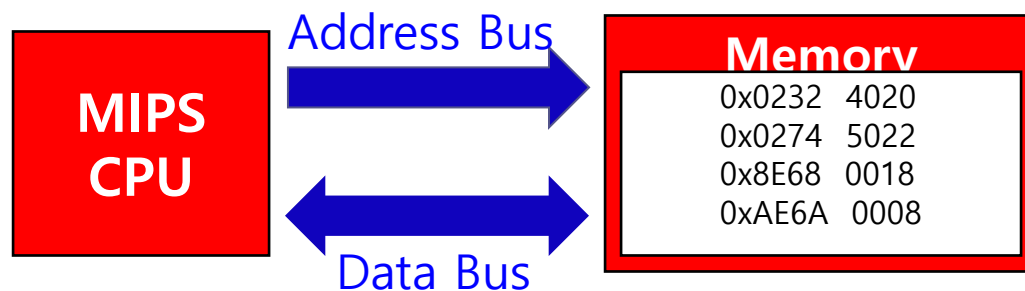
- Check out the class web for instructions on how to set up an environment to generate the MIPS code using Eclipse on Windows
- Test-generate binary from the MIPS assembly program with assembler

```
add    $t0, $s1, $s2 # $t0 <= $s1 + $s2
sub     $t2, $s3, $s4 # $t2 <= $s3 - $s4
lw      $t0, 24($s3)  #load (read) word from memory
                        # $t0 <= [$s3 + 24]
sw      $t2, 8($s3)   # store(write) word to memory
                        # [$s3 + 8] <= $t2
```

assembler

0x0232	4020
0x0274	5022
0x8E68	0018
0xAE6A	0008

Don't worry. We are going to talk deep about this!



MIPS Instruction examples in 2 forms

- Human-readable form

```
addi $2, $0, 5      // $2 = $0 + 5
sub  $7, $7, $2      // $7 = $7 - $2
and  $5, $3, $4      // $5 = $3 & $4
```

- Machine-readable form

```
0010 0000 0000 0010 0000 0000 0000 0101 = 0x20020005 // addi $2, $0, 5
0000 0000 1110 0010 0011 1000 0010 0010 = 0x00e23822 // sub  $7, $7, $2
0000 0000 0110 0100 0010 1000 0010 0100 = 0x00642824 // and  $5, $3, $4
```