Lecture 03

제어 흐름

리뷰

- 변수
  - 시스템 메모리에 저장되는 값 혹은 계산할 수 있는 표현과 연계되는 이름
- 데이터형
  - 해당 변수는 메모리에 어떤 형식으로 저장되고, 어떤 값으로 할당되고, 어떻게 처리되어야 할 지를 명시적으로 알려줌
- 연산자
  - 해당 변수는 어떻게 처리되어야 할 지를 알려줌
  - 3가지 종류(unary, binary, ternary)가 있음
- 수식 표현
  - 상수, 변수, 연산자, 함수 등을 포함하는 프로그램문

리뷰

- 데이터형
  - 기본적인 데이터형
    숫자형(**int**, **float**, **double**)
    문자형(**char**)
    자료형(**struct**, **union**)
  - 크기

sizeof(**char**) < **sizeof**(**short**) ≤ **sizeof**(**int**) ≤ **sizeof**(**long**) and
sizeof(**char**) < **sizeof**(**short**) ≤ **sizeof**(**float**) ≤ **sizeof**(**double**)

  - 부호형(**signed**) 및 무부호형(**unsigned**)
  - 메모리에 저장되는 방식: big endian 및 little endian
  - **형변환 유의**
    보통 계산을 진행하기 전에 “작은 ” 쪽에서 “큰” 쪽으로 변환됨

리뷰

- 연산자
  - 오퍼랜드 개수에 따른 종류
    - Unary : `-, ++`
    - Binary : `+, -, *, /, %`
    - Ternary : `?:`
  - 기능에 따른 종류
    - 산술 : `+ - * / %`
    - 관계 : `> >= < <= == !=`
    - 논리 : `&& || !`
    - 비트 : `& | ^ << >>`
    - 증가/감소/지정 : `++ -- += -= *= /= %= <<= >>= &= |= ^=`
- 조건문 **`if else`**
- 우선순위 및 계산순서

문장과 블록

- 수식(expression)
  - `x=0, y=x+2, y++, printf("hello world")` 등
- 문장(statement)
  - 수식이 세미콜론(;)으로 끝나는 것
  - `x=0; y=x+2; y++; print("hello world");` 등
- 복합문(compound statement)/블록
  - 중괄호 {}로 묶인 여러 개의 선언문이나 문장

```
int main() {
    char msg[] = "hello world";
    puts(msg);
    return 0;
}
```

복합문/블록

블록

- 구문상으로 단일문장과 동일한 기능을 수행함
- 블록 속에 변수를 선언할 수 있음

```
{
    int a=10;
    int y=square(a);
}
```

- 비어 있는 블록을 사용할 수 있음

```
{}
```

- 블록의 끝에 세미콜론이 없음

중첩된 블록

- 블록이 다른 블록 속에 들어갈 수 있음

```
int y=0;
{
   int a=10;
   y=square(a);
   {
      float b=3.567;
      y += square_root(b);
   }
}
```

제어 조건

- C++나 Java와 달리 C89/C90에는 boolean 자료형이 없음
  - C99에는 `stdbool.h` 표준 라이브러리에서 boolean 자료형이 제공됨
- 조건은 당일 수식이나 논리 연산자로 연결된 수식을 말함
  - 예, `x<3` 혹은 `x<y && x>0`
  - 수식은 0이 아닌 값으로 계산되면 조건이 참(true)이며, 0인 값으로 계산되면 거짓(false)임
  - 수식은 숫자형이나 포인터가 되어야 함

```
const char msg[] = "some text";
if (msg) // msg가 가리키는 문자는 null 문자가 아닌지 확인
    return 0;
```

제어 흐름 – 조건문

- **`if else`** 문
- **`else if`** 문
- **`switch`** 문

if else 문

- 일반적인 포맷

```
if (수식)
    블록 1
else
    블록 2
```

- 수식이 먼저 계산되고, 계산 값이 참이면 블록 1이 수행되며, 계산 값이 거짓이면 블록 2가 수행됨
- **else** 부분은 필요에 따라 생략 가능
- 블록 속에 문장이 하나이면 중괄호 생략 가능

if else 문

- 예,

```
if (x%2)
    y += x/2;
else
    y += (x+1)/2;
```

  - 조건을 검사하기
    수식 x%2는 계산되며 반환되는 값을 검사하기
  - 검사 결과는 참이면 **if** 밑에 문장이 수행됨
    y += x/2는 수행됨
  - 검사 결과는 거짓이면 **else** 밑에 문장이 수행됨
    y += (x+1)/2는 수행됨

if else 문

- 예, **else** 부분이 생략될 때

```
if (x%2)
    y += x/2;
```

  - 조건을 검사하기
    수식 x%2는 계산되며 반환되는 값을 검사하기
  - 검사 결과는 참이면 **if** 밑에 문장이 수행됨
    Y += x/2는 수행됨
  - 검사 결과는 거짓이면 아무 문장이 수행되지 않음

else if 문

- 일반적인 포맷

```
if (수식 1)
    블록 1
else if (수식 2)
    블록 2
else if (수식 3)
    블록 3
else
    블록 4
```

- 수식은 **순서에 의해** 계산되며, 수식이 참일 경우 바로 그 밑에 블록이 수행되며, 그렇지 않을 경우 다음 수식으로 넘어가는 과정을 반복함
- 수식이 참일 경우가 하나도 없을 때 마지막 **else** 밑에 블록이 수행됨

else if 문

- x=20이면 y 값이 얼마 인가?
- x=25이면 y 값이 얼마 인가?
- x=26이면 y 값이 얼마 인가?

```
if (x%5==0)
   if (x%2==0)
      y=1;
else
   y=2;
```

else if 문

- x=26인 경우 y=2가 되도록 하려면 중괄호를 사용함

```
if (x%5==0){
    if (x%2==0)
        y=1;
} else
    y=2;
```

switch 문

- 일반적인 포맷

```
switch (수식) {
    case 상수:
        문장
        break;
    case 상수:
        문장
        break;
    default :
        문장
        break;
}
```

- 수식은 계산되며, 그 결과가 어떤 상수와 일치하면 바로 뒤의 문장은 수행됨
- **case** 다음에 오는 상수들은 반드시 달라야 하고, 정수 값을 가져야 함
- **default**는 각 경우가 만족되는 것이 없을 때 수행되고, 생략해도 됨

switch 문

- 출력이 무엇인가?

```
char ch='a';
switch (ch){
   case 'A': printf("case 1\n"); break;
   case 'a': printf("case 2\n"); break;
   case 'b': printf("case 3\n"); break;
   default: printf("default\n"); break;
}
```

switch 문

- **break**가 없으면 출력이 어떻게 바꾸는가?

```
char ch='a';
switch (ch){
   case 'A': printf("case 1\n");
   case 'a': printf("case 2\n");
   case 'b': printf("case 3\n");
   default: printf("default\n");
}
```

- **break**가 없으면 다음 **case**로 넘어감

제어 흐름 – 순환문

- `while` 문
- `for` 문
- `do while` 문
- `break` 및 `continue`

while 문

- 일반적인 포맷

```
while (수식)
    블록
```

- 수식을 먼저 계산하며, 결과가 0이 아니면 블록이 수행되며, 수식을 다시 계산하게 됨
- 수식이 0이 되면 블록이 수행되지 않음
- 수식을 항상 먼저 계산하니 블록이 한번도 수행되지 않을 수 있음

for 문

- 일반적인 포맷

```
for (수식 1; 수식 2; 수식 3)
    블록
```

- 수식 1, 수식 2, 수식 3이 모두 수식이어야 함
- 일반적으로
  - 수식 1과 수식 3은 **지정문**(assignment)이거나 **함수호출**임
  - 수식 2는 **관계수식**(relational expression)임
- 3개의 수식 중 어느 것이라도 생략은 가능하나 세미콜론은 남겨 두어야 함

for 문

- 예, 계승을 계산하는 함수

```
int factorial(int n) {
    int i, f=1;
    for (i=1; i<=n; i++)
        f *= i;
    return f;
}
```

수식 1: `i=1`
수식 2: `i<=n` (**for** 문을 계속 돌리는 조건)
수식 3: `i++` (`i` 값을 증가시켜야 **for** 문을 벗어날 수 있음)

for 문

- **for** 문이 다음의 **while** 문과 같음

```
수식 1;
while (수식 2) {
    블록
    수식 3;
}
```

```
int factorial(int n) {
    int i, f=1;
    for (i=1; i<=n; i++)
        f *= i;
    return f;
}
```

```
int factorial(int n) {
    int i=1, f=1;
    while (i<=n) {
        f *= i;
        i++;
    }
    return f;
}
```

do while 문

- **while**이나 **for** 문은 종료 조건을 먼저 검사하니 프로그램은 수행되지 않을 수 있음
- **do while** 문은 프로그램은 최소한 한 번은 수행됨

```
do
    블록
while (수식);
```

블록이 수행된 후 수식이 계산되며, 결과가 참이면 블록이 다시 수행되며, 거짓이면 루프를 종료하게 됨

do while 문

- 예,

```
int factorial(int n) {
    int i=1, f=1;
    do {
        f *= i;
        i++;
    } while (i<=n);
    return f;
}
```

break 및 continue

- **break** 문은 가장 가까운 루프 하나를 벗어나며, 중첩된 루프일 경우 한 번에 전체 루프에서 빠져나올 수 없음

```
char c;
do {
   puts("Keep going (y/n): ");
   c = getchar();
   if (c != 'y')
      break;
} while (c != '\n');
```

입력된 문자가 'y'가 아니면 **do while**을 벗어남

break 및 continue

- **break** 문은 **switch**, **while**, **for**, **do while** 문에서 사용할 수 있음
- **continue** 문은 루프 안에서만 사용될 수 있으며, **switch** 문에서 사용될 수 없음
- **continue** 문은 루프에서 다음 반복을 넘기기 위해 사용됨

```
// 짝수의 계승을 계산하는 코드
int i, f=1;
for (i=1; i<=n; i++) {
    if (i%2 != 0) // i가 홀수이면 곱해지지 않음
        continue;
    f *= i;
}
```

함수

- 특정한 패턴을 포함한 행을 출력하는 프로그램을 작성해봄

입력:

Ah love! Could you and I with Fate conspire
To grasp this sorry Scheme of Things entire,
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!

"**ould**"라는 패턴이 있는 행을 출력하면:

Ah love! C**ould** you and I with Fate conspire
W**ould** not we shatter it to bits -- and then
Re-m**ould** it nearer to the Heart's Desire!

함수

- 해결하려는 문제를 더 작은 문제들로 분할함

```
while (행이 있는 동안)
    if (그 패턴이 행 속에 있으면)
        그 행을 출력함
```

- 작은 문제들을 독립적으로 해결함

“행이 있는 동안”	`getline`(입력) 함수
“그 패턴이 행 속에 있으면”	`strindex`(행, 패턴) 함수
“그 행을 출력함”	`printf`(행) 함수

함수

- getline() 함수

```
int getline(char s[], int lim) {
    int c, i=0;
    while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
        s[i++] = c;
    if (c=='\n')
        s[i++] = '\n';
    s[i] = '\0';
    return i;
}
```

함수

- strindex() 함수

```
int strindex(char s[], char t[]) {
    int i, j, k;
    for (i=0; s[i]!='\0'; i++) {
        for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
            ;
        if (k>0 && t[k]=='\0')
            return i;
    }
    return -1;
}
```

함수

- main() 함수

```
#define MAXLINE 1000

char pattern[] = "ould";

int main() {
    char line[MAXLINE];
    while (getline(line, MAXLINE) > 0) {
        if (strindex(line, pattern) >= 0)
            printf("%s", line);
    }
    return 0;
}
```

함수

- 일반적인 포맷

```
리턴형 [space] 함수 이름(매개변수 선언) {
    선언문과 다른 문장들
}
```

- 프로그램은 개별적인 함수의 집합임
- 함수 사이의 연락은 매개변수와 리턴 값으로 이루어지며, 외부변수(**global variable**)를 통해서도 가능함
- **`return`** 문은 호출된 함수로부터 계산된 값을 호출한 함수로 넘겨줌

함수

- 함수들은 소스 파일 하나에 다 있어도 되고, 여러 개의 소스 파일에 나누어 있어도 됨

main.c

```c
#include <stdio.h>
#include "aux_func.h"

int main() {
    …
    return 0;
}
```

aux_func.h

```c
#ifndef __AUX_FUNC_H__
#define __AUX_FUNC_H__

int getline(char s[], int lim);
int strindex(char s[], char t[]);

#endif
```

aux_func.c

```c
#include <stdio.h>
#include "aux_func.h"

int getline(char s[], int lim) {
    …
}

int strindex(char s[], char t[]) {
    …
}
```

컴파일

```
gcc -Wall main.c aux_func.c -o main
```

되부름(recursion)

- 함수는 직접적으로 또는 간접적으로 자기 자신을 호출할 수 있음
- 예, 1부터 n까지의 합

$$\sum(n) = 1 + 2 + 3 + \cdots + n - 1 + n$$

```
int sum(int n) {
    int i, s=0;
    for (i=1; i<=n; i++)
        s += i;
    return s;
}
```

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

되부름(recursion)

- 예, 1부터 n까지의 합

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

```
int sum(int n) {
    if (n==0) return 0; // 종료 조건
    if (n==1) return 1; // 종료 조건
    return n+sum(n-1);  // 되부름
}
```

되부름(recursion)

- 장점
  - 간결하고 명확함
  - 코드 작성 및 이해가 편리함
- 단점
  - 메모리 사용량이 높음
  - 속도가 느림

변수 범위

- 어떤 변수의 범위란 프로그램 중에서 그 변수가 효력을 발생하는 부분임

```
int a=10; // a: 외부변수
…
int factorial(int n) { // n: 매개변수
    int i, f=1; // i, f: 자동변수
    for (i=1; i<=n; i++)
        f *= i;
    return f;
}
```

변수 범위

- 자동변수는 함수 내에서 선언되며, 그 변수의 범위는 함수 내이며, 함수 매개변수가 마찬가지임
- 외부변수의 범위는 소스 파일의 선언되는 부분부터 그 파일의 끝 부분까지임

```
int a=10;

int main() {…}

double b=1.3;

int func1(int k) {int a=2; …}

float func2(int k) {int b=12; …}
```

변수 범위

- 외부변수는 한 소스 파일에서 선언되며, 그 변수를 다른 소스 파일에서 사용하려면 **extern** 선언을 포함해야 함

file1.c

```
#include <stdio.h>

extern char msg[];

int main() {
    printf("%s", msg);
    return 0;
}
```

file2.c

```
…
char msg[] = "hello world";
…
int main() {…}
```

출력: hello world

정적변수

- 변수선언 앞에 **static** 문을 넣으면 그 변수가 정적변수임
  - 외부변수나 함수에서의 **static** 선언문은 그 파일 내에서만 통용되며, **extern** 선언을 할 수 없음
  - 자동변수에서의 **static** 선언문은
    - 함수 내에서만 통용됨
    - 프로그램을 초기화할 때 한 번만 초기화됨
    - 함수가 호출될 때마다 다시 초기화되지 않음

```
void foo() {
    static int a=0;
    printf("%d\n", a++);
    …
}
```

함수호출	a의 값
#1	0 (초기화)
#2	1
#3	2
…	…

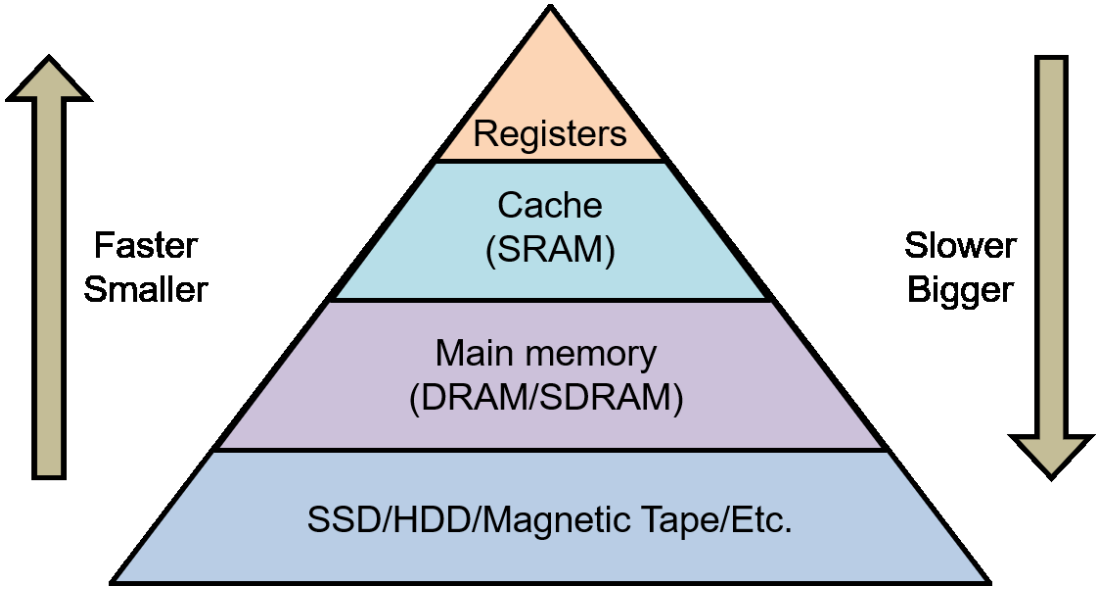
레지스터 변수

- 매우 자주 사용되는 변수들을 레지스터 변수로 선언할 수 있음
- 레지스터 변수는 시스템 메모리에 위치하지 않고, 컴퓨터의 레지스터에 위치함
  - 레지스터 변수는 어드레스를 갖고 있지 않음
  - 레지스터 변수에 가리키는 포인터를 선언할 수 없음
- 자동변수나 함수의 매개변수에만 적용될 수 있음

```
void foo(register int n) {
    register long m;
    register char c;
    …
}
```

레지스터 변수

- 컴퓨터 하드웨어에 따라
  - 특정한 데이터형만 허용됨
  - 레지스터 변수로 선언될 수 있는 개수가 제한됨
    (가능한 개수 이상의 레지스터 변수들을 보통변수로 컴파일됨)



프리프로세서

- 파일 삽입

  #include “파일 이름”

  혹은

  #include <파일 이름>

- **#include** “파일 이름”
  - 사용자 정의 헤더파일을 삽입할 때 사용함
  - 파일 경로를 따로 제공하지 않으면 컴파일러가 소스 파일이 있는 경로나 표준 시스템 경로에 파일을 찾아냄
- **#include** <파일 이름>
  - 표준 라이브러리를 삽입할 때 사용함
  - 컴파일러가 표준 시스템 경로에 파일을 찾아내고 컴파일함
  - 소스 파일이 있는 경로에 찾지 않음

프리프로세서

- 매크로(macro) 치환

  #define 이름 문자열

- 긴 문자열을 하나의 심벌로 대치하여 사용할 수 있음
- 예, 제곱을 계산하는 매크로

```
#define square(x) (x)*(x)
```

소스 코드에서

```
int main() {
    …
    int z = square(10); // 컴파일 시 int z = (10)*(10);로 대치됨
}
```

다음이 맞는가?

```
#define square(x) x*x
```

프리프로세서

- 조건부 포함
  - 컴파일 시 조건에 따라 포함할지 포함하지 않을지를 선택할 수 있음
  - **#if**, **#elif**, **#else**, **#ifdef**, **#ifndef**, **#endif** 명령 사용 가능
  - 조건은 정수형 표현(**sizeof**, **casts**, **enum** 제외)이 되어야 함

```
#ifndef CONTROLLER
#define CONTROLLER

/* controller.h 내용 */

#endif
```

```
#if SYSTEM == MAC
  #define CONTROLLER "con_mac.h"
#elif SYSTEM == WIN
  #define CONTROLLER "con_win.h"
#elif SYSTEM == UNIX
  #define CONTROLLER "con_unix.h"
#else
  #define CONTROLLER "default.h"
#endif

#include CONTROLLER
```