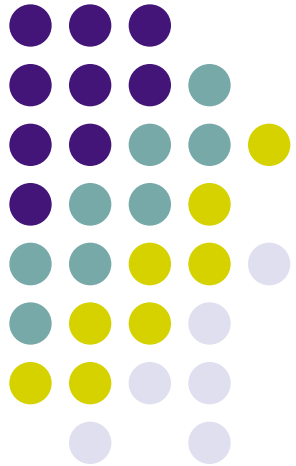
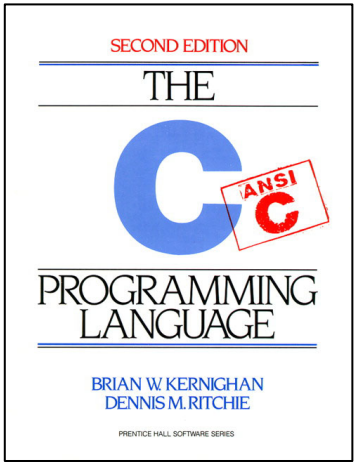
Intro to C

2019. Spring



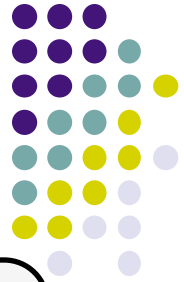
C



- Created in 1972 by Dennis Ritchie
  - Designed for creating system software
  - Portable across machine architectures
  - Most recently updated in 1999 (C99) and 2011 (C11)

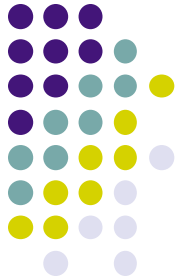
- Characteristics
  - “Low-level” language that allows us to exploit underlying fe atures of the architecture – but easy to fail spectacularly (!)
  - Procedural (not object-oriented)
  - “Weakly-typed” or “type-unsafe”

Generic C Program Layout



```c
#include <system_files>
#include "local_files"

#define macro_name macro_expr

/* declare functions */
/* declare external variables & structs */

int main(int argc, char* argv[]) {
  /* the innards */
}

/* define other functions */
```

C Syntax: main (1/2)

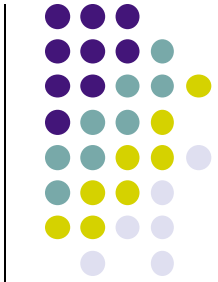


- To get command-line arguments in main, use:
  - int **main**(int argc, char* argv[])

```
int main(int argc, char** argv)
```

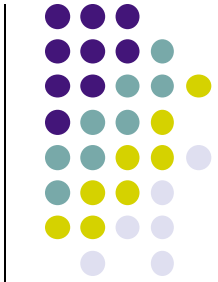
- What does this mean?
  - argc contains the number of strings on the command line (t he executable name counts as one, plus one for each argu ment).
  - argv is an array containing *pointers* to the arguments as stri ngs (more on pointers later)

C Syntax: main (2/2)



- Example: `$ foo hello 87`
  - `argc = 3`
  - `argv[0]="foo", argv[1]="hello", argv[2]="87"`

Primitive Types in C

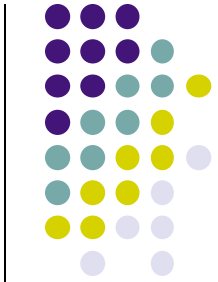


- Integer types
  - char, int
- Floating point
  - float, double
- Modifiers
  - short [int]
  - long [int, double]
  - signed [char, int]
  - unsigned [char, int]

C Data Type	32-bit	64-bit	printf
char	1	1	%c
short int	2	2	%hd
unsigned short int	2	2	%hu
int	4	4	%d / %i
unsigned int	4	4	%u
long int	4	8	%ld
long long int	8	8	%lld
float	4	4	%f
double	8	8	%lf
long double	12	16	%Lf
pointer	4	8	%p

Typical sizes – see `sizeofs.c`

Basic Data Structures (1/2)



- C does not support objects!!!

- **Arrays**
  - contiguous chunks of memory
  - Arrays have no methods and do not know their own length
  - Can easily run off ends of arrays in C - security bugs!!!

Basic Data Structures (2/2)

- **Strings**
  - null-terminated char arrays
  - Strings have no methods, but `string.h` has hel pful utilities

```
char* x = "hello\n";
```

- **Structs**
  - the most object-like feature, but are just collecti ons of fields

Function Ordering

- You *shouldn't* call a function that hasn't be en declared yet

sum_badorder.c

```c
#include <stdio.h>

int main(int argc, char** argv) {
  printf("sumTo(5) is: %d\n", sumTo(5));
  return 0;
}

// sum of integers from 1 to max
int sumTo(int max) {
  int i, sum = 0;

  for (i = 1; i <= max; i++) {
    sum += 1;
  }
  return sum;
}
```

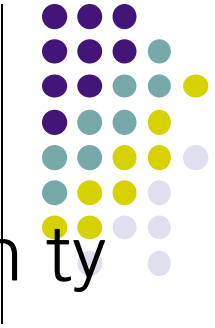
Solution 1: Reverse Ordering

- Simple solution;
  - however, imposes ordering restriction on writin g functions (who-calls-what?)

sum_betterorder.c

```c
#include <stdio.h>

// sum of integers from 1 to max
int sumTo(int max) {
  int i, sum = 0;

  for (i = 1; i <= max; i++) {
    sum += 1;
  }
  return sum;
}

int main(int argc, char** argv) {
  printf("sumTo(5) is: %d\n", sumTo(5));
  return 0;
}
```

Solution 2: Function Declaration

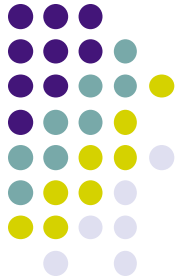


- Teaches the compiler arguments and return types;
  - function definitions can then be in a logical order

sum_declared.c

```c
#include <stdio.h>

int sumTo(int);  // func prototype

int main(int argc, char** argv) {
  printf("sumTo(5) is: %d\n", sumTo(5));
  return 0;
}

// sum of integers from 1 to max
int sumTo(int max) {
  int i, sum = 0;
  for (i = 1; i <= max; i++) {
    sum += 1;
  }
  return sum;
}
```

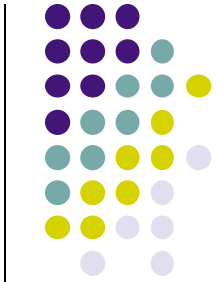
Function Declaration vs. Definition(1/2)



- C/C++ make a careful distinction between these two

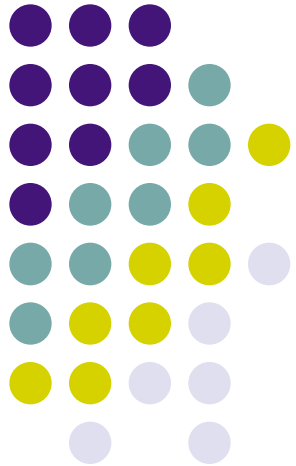
- Declaration: description of a thing
  - *e.g.* function prototype, external variable declaration
    - Often in header files and incorporated via `#include`
    - Should also `#include` declaration in the file with the actual definition to check for consistency
  - Needs to appear in **all files** that use that thing
    - Should appear before first use

Function Declaration vs. Definition(2/2)

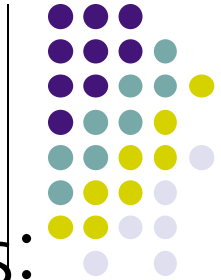


- Definition: the thing itself
  - *e.g.* code for function, variable definition that creat es storage
  - Must be **exactly one** definition of each thing (no d uplicates)

Scope Rules and Storage Types



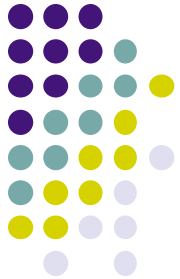
Variable Storage Classes



- Storage class of a variable determines its:
  - Scope
  - Lifetime

- C storage classes are:
  - auto
  - static
  - register
  - extern

Data Storage in Memory

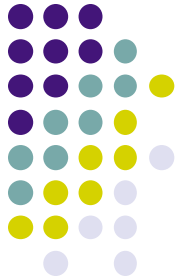


- Variables may be *automatic* or *static*
- *Automatic variables* may *only* be declared *with in* functions and compound statements (*blocks*)
  - Storage *allocated* when function or block is entered
  - Storage is *released* when function returns or block exits
- Parameters and result are (somewhat) like auto matic variables
  - Storage is *allocated* and *initialized* by *caller* of function
  - Storage is *released* after function *returns* to caller.

Scope

- Identifiers declared within a function or compound statement are visible *only* from the point of declaration to the end of that function or compound statement.
  - Like *Java*

Example



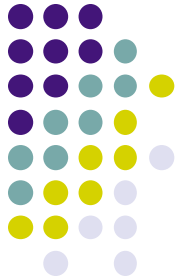
```
int fcn (float a, int b) {

    int i;
    double g;

    for (i = 0; i < b; i++)
    {
            double h = i*g;
            loop body – may access a, b, i, g, h
    }       // for(i…)

     fcn body – may access a, b, i, g

}  //       end of fcn( … )
```

`i` is visible from this point to end of **`fcn`**

`g` is visible from this point to end of **`fcn`**

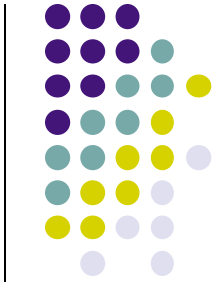
`h` is only visible from this point to end of loop!

Idiosyncrasies



- In traditional *C* & *Visual Studio*
  - *All variables* must be declared at *beginning* of fun ction or compound statement (i.e., before first *stat ement*); visible from that point on
- In **gcc**
  - *Variables* may be declared anywhere in function o r compound statement; visible from that point on
- In *C99* & *C++*
  - *Loop variables* may be declared in **for** statement; visible only to end of loop body, but not beyond

External and static variables



- External variable
  - declared outside the body of a function
- File scope
  - visible from the point of the declaration to the end of the file.
- Static storage duration
  - through the duration of the program.

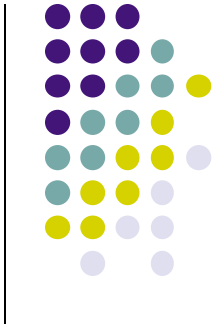
- External/global variables have file scope and static storage duration

static variables

```
static int i;

void f(void)
{
        static int j;
}
```

- static used outside a block
  - the variable is only visible in the file in which it is d eclared
- static used in a block
  - the variable lives beyond the duration of the block, and is initialized only once.

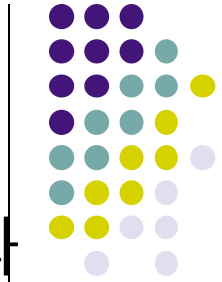
Static Variable Examples



```
#include <stdio.h>

int nextvalue()
{
        static int i = 0;
        i++;
        return i;
}

int main()
{
        int i;
        for(i = 10; i > 0; i--)
        {
                printf("%d\n", nextvalue());
        }
        return 0;
}
```

Output: 결과는?

프로그램 예제

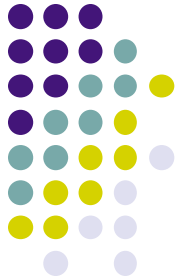


- 사용자가 ID나 비밀번호를 잊어버렸을 경우 자신의 ID와 비밀번호를 조회할 때 사용할 수 있는 코드의 일부만 포함한 보기용 프로그램
  - 프로그램 Tip
    - main 함수는 사용자에게 사용 가능한 서비스 종류를 메뉴로 표시하고 사용자가 종료하기를 선택하지 않는 한 서비스를 계속 이용하도록 함
    - 사용자 ID 조회 서비스는 find_ID 함수로 정의 ID 조회 횟수 count_ID는 정적 지역 변수로 선언 PW 조회 횟수 count_PW는 지역 변수로 선언
  - 학습 point
    - 지역 변수와 정적 지역 변수의 차이를 알 수 있음
    - 정적 지역 변수인 count_ID는 find_ID 함수 내에서만 사용 가능하며, 이전 호출 결과가 유지됨을 알 수 있음

stlocal.c (1/2)

```
# include <stdio.h>

void find_PW();
void find_ID();

int main()
{
    int service;                    // 사용자가 요청한 서비스 번호 저장
    do
    {
            printf("₩n <<<1. ID 찾기 2. PW 찾기 3. 종료하기 >>>₩n");
            printf(" 원하는 서비스 번호를 입력하세요(1-3)");
            scanf("%d", &service);

            switch(service)
            {
                    case 1: find_ID(); break;       // 아이디 찾기 서비스 진행
                    case 2: find_PW();              // 비밀번호 찾기 서비스 진행
            }
    } while (service != 3);         / 3. 종료하기를 선택하지 않은 한 계속 반복
    return 0;
}
```

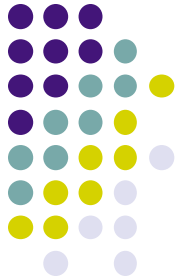
stlocal.c (2/2)



```
void find_ID()
{
    static int count_ID=0;                    // 이 함수의 호출 횟수 저장

    count_ID++;
    printf("₩n 총 %d회째 ID 찾기 요청입니다. ₩n",count_ID);
}

void find_PW()
{
    int count_PW=0;                           // 자동(지역) 변수
    count_PW++;
    printf("₩n 총 %d회째 비밀번호 찾기 요청입니다. ₩n",count_PW);
}
```

정적 지역 변수

정적 전역 변수



- 함수 밖에서 선언하되 앞에 static을 붙임
  - 프로그램 전체에서 참조 가능
- 지속 기간은 전역 변수와 같이 프로그램 시작~끝
- 자동으로 0으로 초기화됨
- 전역 변수와의 차이점
  - 한 프로그램이 여러 소스 파일로 나눠진 경우
    - 전역 변수: 프로그램 전체에서 참조 가능 즉 다른 소스 파일에서도 참조 가능
    - 정적 전역 변수: 선언한 소스 파일에서만 참조 가능

stglobal.c (1/2)

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

char title[10]="Quiz"; // 게임명

static int count_service = 0;          // 총 서비스 요청 횟수

int main()
{
    int service;    // 사용자가 요청한 서비스 번호 저장
    do
    {
            printf("₩n <<<1. ID 찾기 2. PW 찾기 3. 종료하기 >>>₩n");
            printf(" 원하는 서비스 번호를 입력하세요(1-3)");
            scanf("%d", &service);

            switch(service)
            {
                    case 1: find_ID(); break;          // 아이디 찾기 서비스 진행
                    case 2: find_PW();                 // 비밀번호 찾기 서비스 진행
            }
    } while (service != 3);          /// 3. 종료하기를 선택하지 않은 한 계속 반복
    return 0;
}
```

다른 파일에서 외부 변수로 선언하여 참조할 수 있는 전역 변수를 선언함

현재 파일에서만 참조할 수 있는 정적 전역 변수를 선언함

main 함수에서만 참조할 수 있는 지역(자동) 변수를 선언함

stglobal.c (2/2)

```
void find_ID()
{
   static int count_ID=0;   // 이 함수의 호출 횟수 저장

   count_ID++;                    // find_ID 함ㅅ가 호출될 때 마자 1 증가
   count_service++;                        // 전체 서비스 요청 횟수를 1증가

    printf("₩n %s 게임 방문 중₩n",title);
   printf("₩n 총 전체 서비스 요청 %d번 중 %d회째 ID 찾기 요청입니다. ₩n",
        count_service, count_ID);
}

void find_PW()
{
   static int count_PW=0;      // 정적 변수

   count_PW++;                    //find_PW 함수가 호출될 댸 1증가
   count_service++;                        // 전체 서비스 요청 횟수 1 증가

   printf("₩n %s 게임 방문 중₩n",title);
   printf("₩n 총 전체 서비스 요청 %d번 중 %d회째 PW 찾기 요청입니다. ₩n",
         count_service, count_PW);
}
```

이 함수에서만 참조할 수 있으며 이전 호출 결과 값이 유지되는 정적 지역 변수를 선언함

이 함수에서만 참조할 수 있으며 이전 호출 결과 값이 유지되는 정적 지역 변수를 선언함

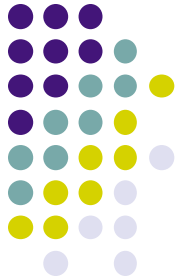
Extern Variables

```
/* File name: file1.c */
extern int i;

void f(void) {
  i++;
}
```

```
/* File name: sum.h */
int i = 0;
extern void f(void);

void g(void) {
  f();
  printf("%d\n", i);
}
```

- informs the compiler that i is an int variable, but does n't cause it to allocate space.

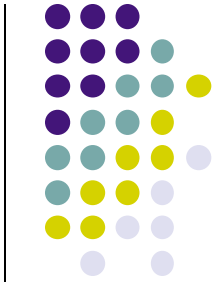
Extern Variables (continued)



- Examples:
  - **`stdin`**, **`stdout`**, **`stderr`** are **`extern`** variables th at point to standard input, output, and error str eams.

- **`extern`** variables
  - Frequently occur in **`.h`** files.
  - Each must be actually declared outside any func tion in exactly one **`.c`** file

typedef



- You can define new types using typedef

```
typedef unsigned int size_t;
```

- Another example

```
typedef struct Books {
   char title[50];
   char author[50];
   char subject[100];
   int book_id;
} Book;

int main()
{
   Book book;
   strcpy( book.title, "C Programming");
   strcpy( book.author, "Nuha Ali");
   strcpy( book.subject, "C Programming Tutorial");
   book.book_id = 6495407;
   ...
}
```

Header files

- When you begin to split up your C program into multiple files
  - you need header files to store function and type declarations

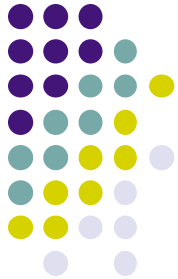
main.c

```
void add(int);
int isEmpty();
extern List *head;

int main()
{
  add(10);
  isEmpty();
  head = NULL;
}
```

list.c♪

```
List *head = NULL;

int isEmpty()
{...}
void add(int v)
{...}
void remove(int v)
{...}
```

Good!



list.h

```
struct node {
  int value;
  struct node * next;
} ;

typedef struct node List;
extern List *head;

int isEmpty(int);
void add(int);
void remove(int)
```

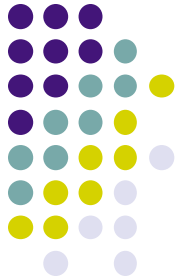
main.c

```
#include "list.h"

int main()
{
  add(10);
  isEmpty();
  head = NULL;
}
```

list.c

```
#include "list.h"

List *head = NULL;

int isEmpty()
{...}
void add(int v)
{...}
void remove(int v)
{...}
```

Wrong



list.h

```
struct node {
  int value;
  struct node * next;
} ;

typedef struct node List;
List *head = NULL;

int isEmpty(int);
void add(int);
void remove(int)
```

Wrong!

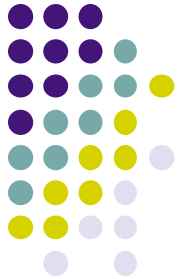
main.c

```
#include "list.h"

int main()
{
  add(10);
  isEmpty();
}
```

list.c

```
#include "list.h"

int isEmpty()
{...}
void add(int v)
{...}
void remove(int v)
{...}
```

Wrong



Wrong!

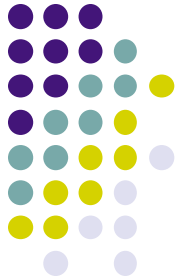
main.c

```
#include "list.c"

int main()
{
  add(10);
  isEmpty();
}
```

list.c

```
List *head = NULL;

int isEmpty()
{...}
void add(int v)
{...}
void remove(int v)
{...}
```

Another good one



list.h

```
struct node {
  int value;
  struct node * next;
} ;

typedef struct node List;

int isEmpty(List *, int);
void add(List * , int);
void remove(List * , int)
```

Allows more than one List to be used

main.c

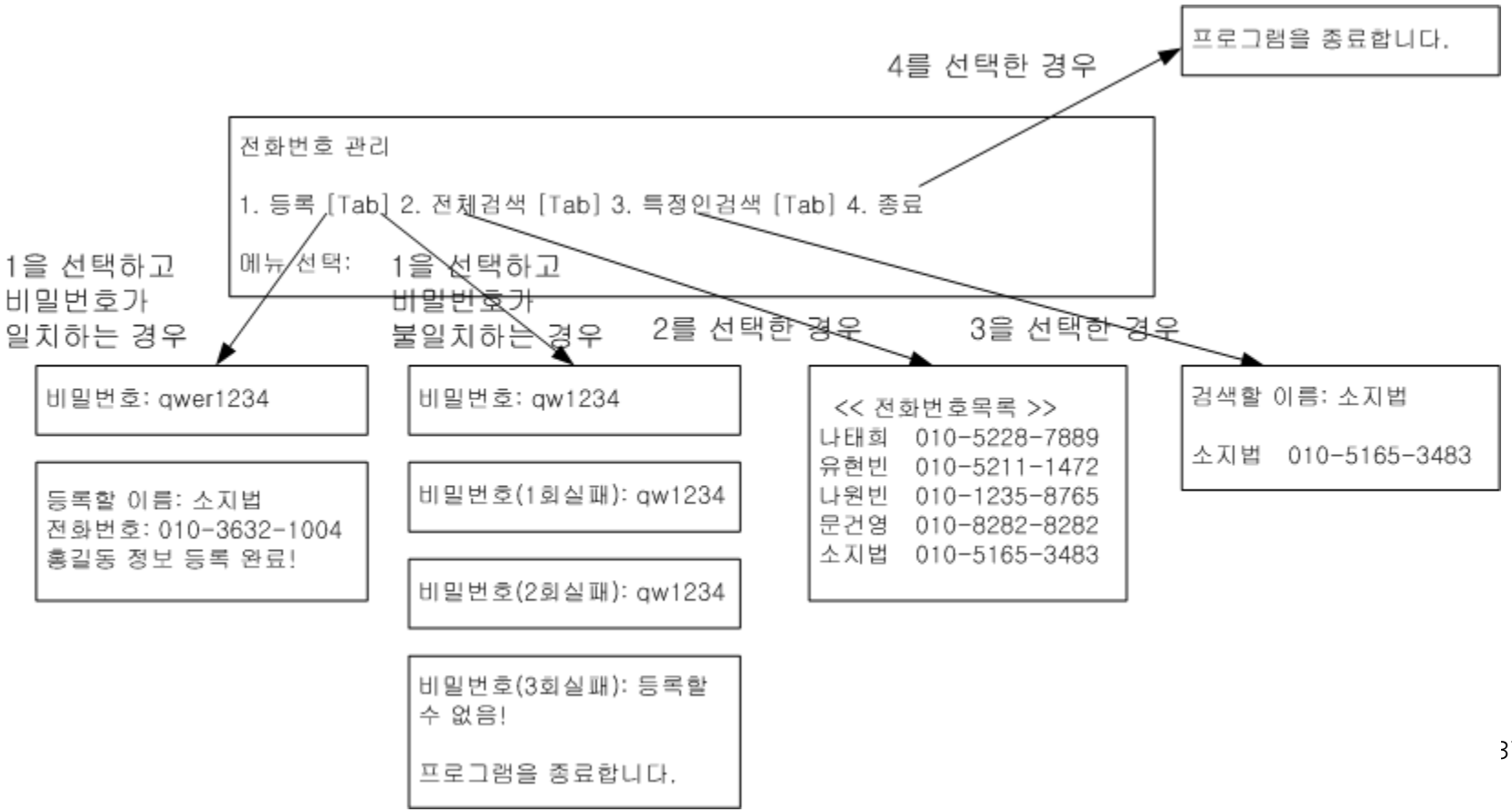
```
#include "list.h"

int main()
{
  List *list1 = NULL;
  add(list1, 10);
  isEmpty(list1);
}
```

list.c

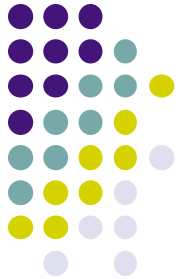
```
#include "list.h"

int isEmpty(List *h)
{...}
void add(List *h, int v)
{...}
void remove(List *h, v)
{...}
```

프로그램 실습 (1/2)

- 다음 그림과 같이 개인 정보 등록, 모든 정보 출력, 특정인의 정보 출력 기능을 가진 간단한 전화번호 관리 프로그램을 작성하시오.



프로그램 실습 (2/2)



- 조건
  - 개인 정보 등록, 모든 정보 출력, 특정인의 정보 출력 기능은 각각 register.c, allprint.c, personalprint.c로 구현함
  - 개인 정보 등록 서비스는 시스템의 비밀 번호(qwer1234) 를 알고 있을 때만 가능함
  - 무허가 사용을 막기 위해 비밀번호 오류가 전체 3회 발생하면 개인 정보 등록 서비스를 전면 차단함
- 프로그램 Tip
  - 외부 변수를 사용하여 파일간 데이터를 공유함
  - 정적 변수를 이용해 비밀번호 오류 횟수를 제한하는 프로그램을 작성함

Questions?