Lesson 15
Review

Trịnh Thành Trung

trungtt@soict.hust.edu.vn

C Language Structure

- The first C program (hello.c)

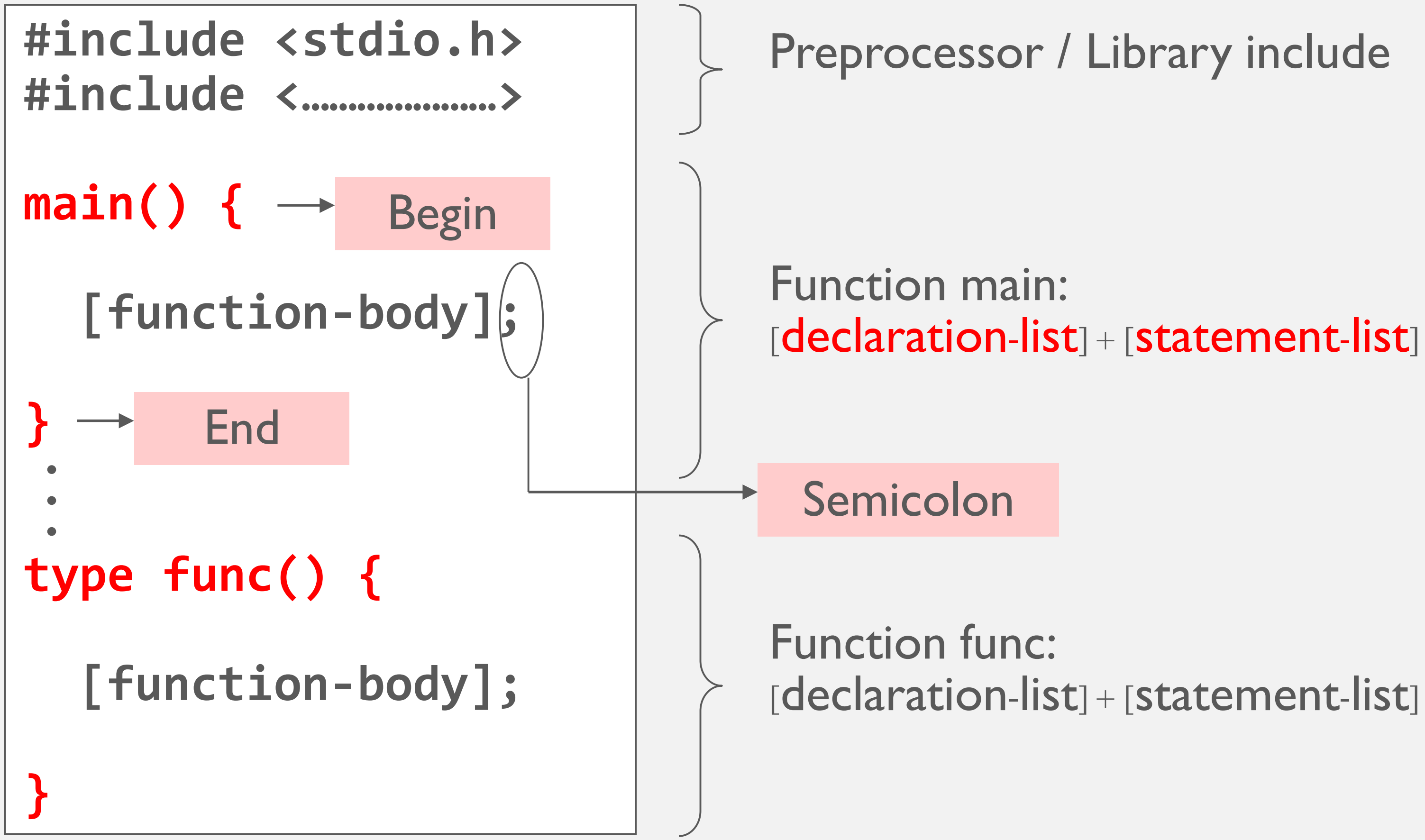
```c
#include <stdio.h>
int main() {
  printf("Hello CP\n");
  return 0;
}
```

Basic gcc examples

- `gcc hello.c` (compile hello.c produce executable a.out)
- `gcc -o hello hello.c` (compile hello.c produce executable hello)
- `gcc -o hello hello.c other.c` (compile hello.c and other.c produce executable hello)

C Language Structure

- General format



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

main() {

   [function-body];

}
 .
 .
 .
type func() {

   [function-body];

}
```

Preprocessor / Library include

Function main:
[declaration-list] + [statement-list]

Function func:
[declaration-list] + [statement-list]

Characteristics of Functions

```
return-type name(argument-list)
{
    local-declarations
    statements
    return return-value;
}
```

- When invoking a function call, we can include function parameters in the parameter list.
- Declaring a function parameter is accomplished by simply including the prototype of the function in the parameter list

Variables declaration

Declaration:

- Tells compiler about variables and their type

Syntax

<typename> varname;

e.g:

```
int i;
float x, y, z;
char c;
```

Assignment: <varname> = <value>;

```
i = 4;
x = 5.4;
y = z = 1.2;
```

Formatting Output with printf

- `printf`
  - precise output formatting
    - Conversion specifications: flags, field widths, precisions, etc.
  - Can perform rounding, aligning columns, right/left justification, inserting literal characters, exponential format, hexadecimal format, and fixed width and precision
- Format

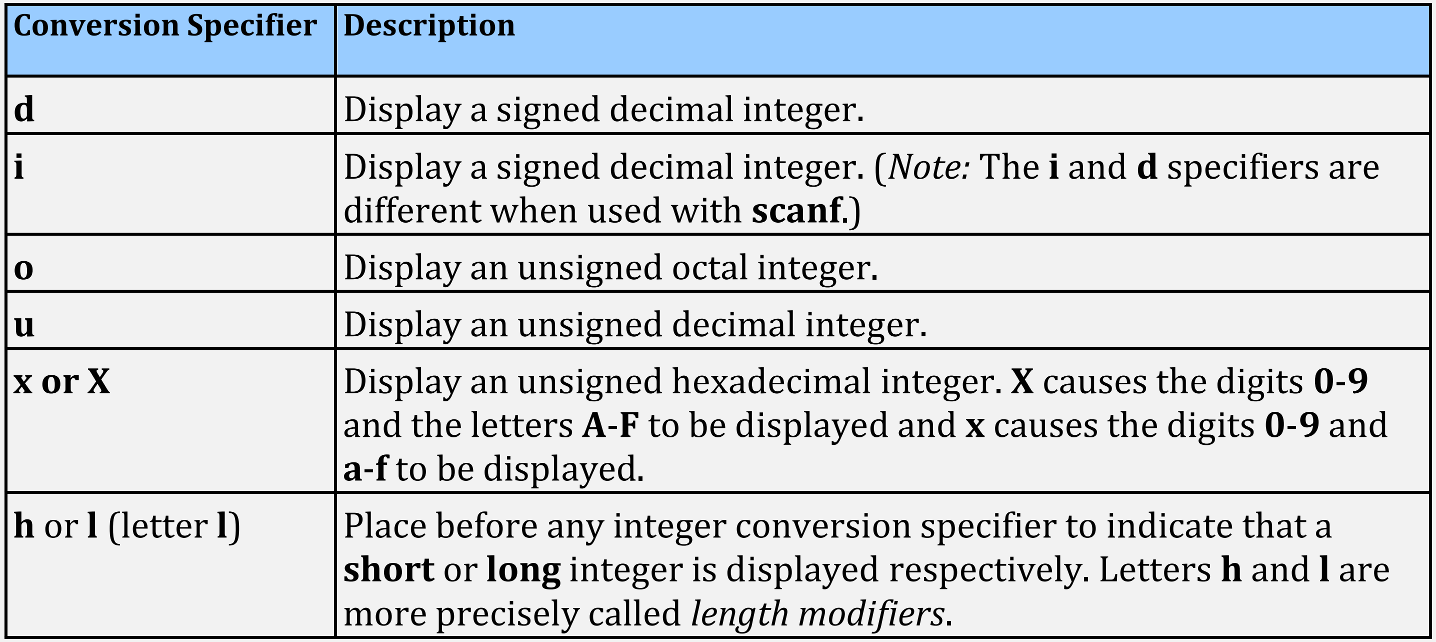
```
printf( format-control-string, other-arguments );
```

  - format control string: includes a listing of the data types of the variables to be output and, optionally, some text and control character(s).
  - other-arguments: correspond to each conversion specification in format-control-string
    - each specification begins with a percent sign, ends with conversion specifier

Printing Integers

- Integer
  - Whole number (no decimal point): 25, 0, -9
  - Positive, negative, or zero

- Only minus sign prints by default (later we shall change this)



Conversion Specifier	**Description**
d	Display a signed decimal integer.
i	Display a signed decimal integer. (*Note:* The **i** and **d** specifiers are different when used with **scanf**.)
o	Display an unsigned octal integer.
u	Display an unsigned decimal integer.
x or **X**	Display an unsigned hexadecimal integer. **X** causes the digits **0-9** and the letters **A-F** to be displayed and **x** causes the digits **0-9** and **a-f** to be displayed.
h or **l** (letter **l**)	Place before any integer conversion specifier to indicate that a **short** or **long** integer is displayed respectively. Letters **h** and **l** are more precisely called *length modifiers*.

Formatting Input with *scanf*

- **`scanf`**
  - Input formatting
  - Capabilities
    - Input all types of data
    - Input specific characters
    - Skip specific characters
- Format

```
scanf(format-control-string, other-arguments);
```

  - format-control-string - describes formats of inputs
  - other-arguments - pointers to variables where input will be stored
  - can include field widths to read a specific number of characters from the stream

Formatting Input with `scanf` (II)

Conversion specifier	Description
Integers	
`d`	Read an optionally signed decimal integer. The corresponding argument is a pointer to integer.
`i`	Read an optionally signed decimal, octal, or hexadecimal integer. The corresponding argument is a pointer to integer.
`o`	Read an octal integer. The corresponding argument is a pointer to unsigned integer.
`u`	Read an unsigned decimal integer. The corresponding argument is a pointer to unsigned integer.
`x` or `X`	Read a hexadecimal integer. The corresponding argument is a pointer to unsigned integer.
`h` or `l`	Place before any of the integer conversion specifiers to indicate that a `short` or `long` integer is to be input.
Floating-point numbers	
`e`, `E`, `f`, `g` or `G`	Read a floating-point value. The corresponding argument is a pointer to a floating-point variable.
`l` or `L`	Place before any of the floating-point conversion specifiers to indicate that a `double` or `long double` value is to be input.
Characters and strings	
`c`	Read a character. The corresponding argument is a pointer to `char`, no null (`'\0'`) is added.
`s`	Read a string. The corresponding argument is a pointer to an array of type `char` that is large enough to hold the string and a terminating null (`'\0'`) character—which is automatically added.
Scan set	
[scan characters	Scan a string for a set of characters that are stored in an array.
Miscellaneous	
`p`	Read an address of the same form produced when an address is output with `%p` in a `printf` statement.
`n`	Store the number of characters input so far in this `scanf`. The corresponding argument is a pointer to integer
`%`	Skip a percent sign (`%`) in the input.

Clear buffer when reading data

- Windows environ t:
  ```
  fflush(st
  ```

undefined behaviour

- Work around:
  ```
  void clear_buffer() {
      int ch;
      while ((ch=getchar()) !=‘\n’ && ch!=EOF);

  }
  ```

Expression and Operations

- Arithmetic Operators
  - Addition `+`
  - Subtraction `-`
  - Multiplication `*`
  - Division `/`
  - Modulation `%`
- Logical Operators
  - AND `&&` `(a > 0) && (b > 0)`
  - OR `||` `(a <= 0) || (b <= 0)`
  - Negation `!` `!(a && c)`

Expression and Operations

- Arithmetic Operators
  - Addition `+`
  - Subtraction `-`
  - Multiplication `*`
  - Division `/`
  - Modulation `%`
- Logical Operators
  - AND `&&` `(a > 0) && (b > 0)`
  - OR `||` `(a <= 0) || (b <= 0)`
  - Negation `!` `!(a && c)`

The if/else Selection Structure

- if
  - Only performs an action if the condition is true.
- if/else
  - A different action when condition is true than when condition is false
  - Psuedocode:

```
If student’s grade is greater than or equal to 60
  Print “Passed”
Else
  Print “Failed”
```

  - Note spacing/indentation conventions

C code

```
if ( grade >= 60 )
  printf( "Passed\n");
else
  printf( "Failed\n");
```

The switch Multiple-Selection Structure

- **`switch`**
  - Useful when a variable or expression is tested for all the values it can assume and different actions are taken.
- Format
  - Series of **`case`** labels and an optional **`default`** case

```
switch ( value ){
  case '1’:
    actions
  case '2’:
    actions
  default:
    actions
}
```

  - **`break`**`;` causes exit from structure

The for Repetition Structure

- Format when using **`for`** loops

```
for ( initialization ; loopContinuationTest ; increment )
  statement
```

No semicolon after last expression

Example:

```
for( int counter = 1; counter <= 10; counter++ )
      printf( "%d\n", counter );
```

- Prints the integers from one to ten.

The while,do Repetition Structure

- while Statement
  - The expression is evaluated. If it is true, statement is executed and expression is reevaluated. This cycle continues until expression becomes false.
  -

```
while (expression)
{
  Statement1;
  Statement2;
    ...
}
```

- do-while Statement
  - The do-while, tests at the bottom after making each pass through the loop body; the body is always executed at least once.

```
do {
   statement1;
   statement2;
   …
} while (expression);
```

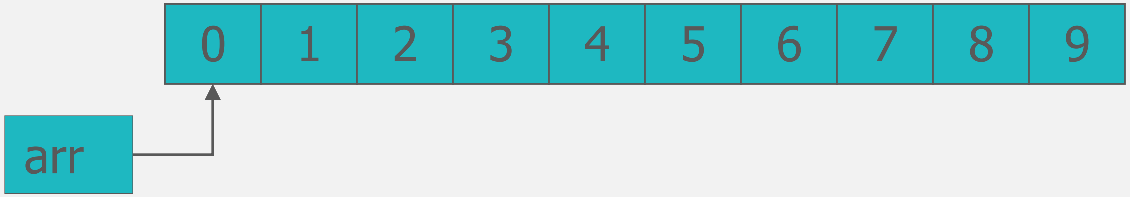
Exercise 15.1

- Create a menu-based program with 3 functions:
  1. Print « Hello » to screen
  2. Ask for user's name then print « Hello *user's name* !» to screen
  3. Exit
- The program will ask for user's input (1-3) then execute the corresponding function. If wrong number's entered, the program will ask user to input again until he enters the right number.

Arrays in Memory

- Sequence of variables of specified type
- The array variable itself holds the address in memory of beginning of sequence
- Example:

```
int arr[10];
```



- The n-th element of array `arr` is specified by `arr[n-1]` **(0-based)**

Multi-dimensional arrays

- Array of arrays:

```
int A[2][3] = { {1, 2, 3},
                {4, 5, 6} };
```

1	2	3
4	5	6

- Means an array of 2 integer arrays, each of length 3.
- Access: j-th element of the i-array is

```
A[i][j]
```

Exercise 15.2

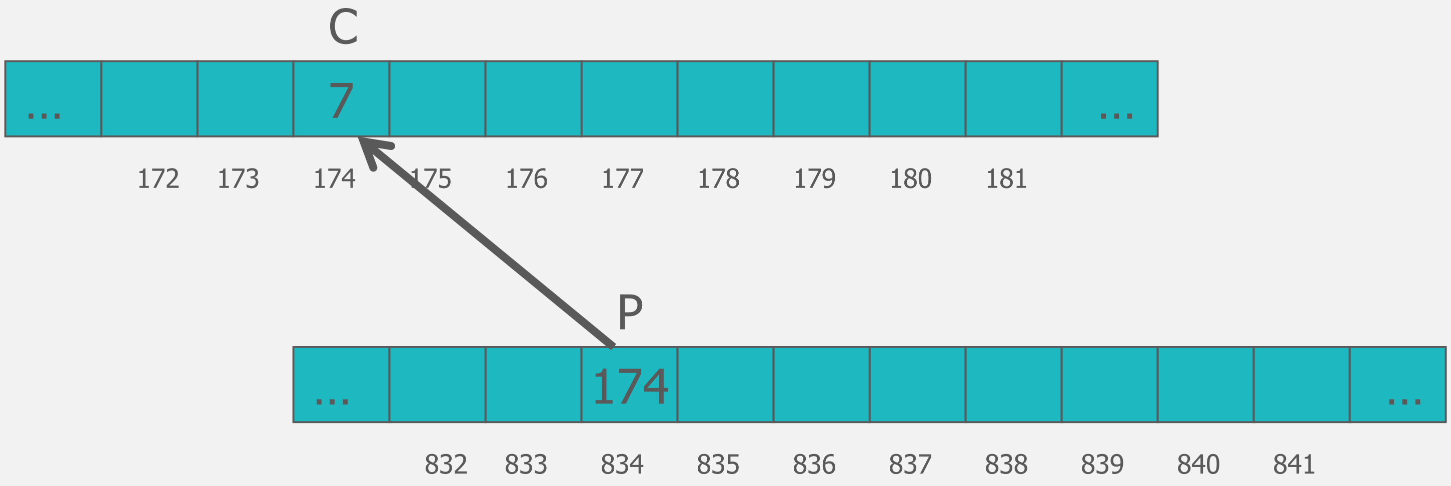
- Add another function to the previous program
  - Ask user to input total number of students
  - Let user input student names, student ids and scores, and store the values in 3 separate arrays
  - Output the student details

```
Student                ID          Score
Nguyen Manh Tuan       20171234    6.5
Vu Thi Huong Giang     20171010    8.5
Hoang Anh Viet         20171100    6.0
```

Declaring a pointer variable

```
type *variable_name;
```

- A pointer is declared by adding a * before the variable name.
- Pointer is a variable that contains an address in memory.
- The address should be the address of a variable or an array that we defined.



Referencing

- The unary operator & gives the address of a variable
- The statement: ptr = &c;

  assigns the address of c to the pointer variable ptr, and now ptr points to c
- To print a pointer, use %p format.

Dereferencing

- The unary operator * is the dereferencing operator
- Applied on pointers
- Access the object the pointer points to
- The statement: *iptr = 5;

  puts in n (the variable pointed to by iptr) the value 5

Pointers and arrays

- Recall that an array S holds the address of its first element S[0]
  - S is actually a pointer to S[0]

```
int s[10];
int *iptr;
iptr=s; /* From now iptr is equivalent to s  */
```

- Both iptr and s now point to s[0]

Pointer arithmetic

- Pointers can be incremented and decremented
- If **p** is a pointer to a particular type, **p+1** yields the correct address of the next variable of the same type
- **p++**, **p+i**, and **p += i** also make sense

Passing arrays to function

- Another way to pass arrays to function is using pointer
- In fact, we pass just the array's address, or more precisely a pointer to the array.
- The function calculate the sum of all array elements.

```
#include <stdio.h>
int addNumbers(int *fiveNumber)
{
    int i,sum=0;
    for(i=0; i<5; i++, fiveNumbers++)
        sum+= *fiveNumbers;
    return sum;
}
```

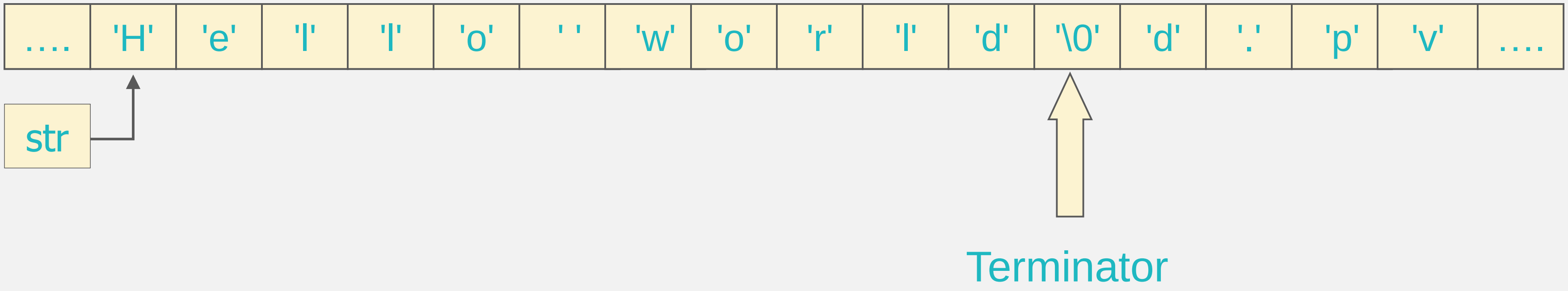
Exercise 15.3

- Modify 15.2 to user pointers instead of arrays

Strings

- An array of characters
- To initialize:

```
char str[] = "Hello World";
```



….	'H'	'e'	'l'	'l'	'o'	' '	'w'	'o'	'r'	'l'	'd'	'\0'	'd'	'.'	'p'	'v'	….

String library

- Use

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

- Functions:

  -
    ```
    strlen(const char s[])
    ```
    returns the length of s

  -
    ```
    strcmp(const char s1[],
           const char s2[])
    ```
    compares s1 with s2

  -
    ```
    strcpy(char s1[],
           const char s2[])
    ```
    copies to contents of s2 to s1
  - …

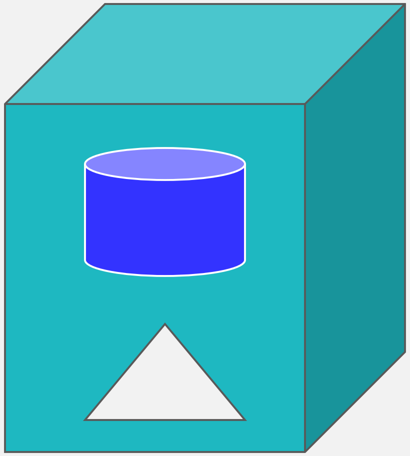
String Conversion Functions

- Conversion functions
  - In **<stdlib.h>** (general utilities library)
- Convert strings of digits to integer and floating-point values

Prototype	Description
double atof(const char *nPtr)	Converts the string **nPtr** to **double**.
int atoi(const char *nPtr)	Converts the string **nPtr** to **int**.
long atol(const char *nPtr)	Converts the string **nPtr** to long **int**.

Structure

- A structure in C is a collection of items of different types.
- Structures, or structs, are very useful in creating data structures larger and more complex than the ones we have discussed so far.



```
struct struct-name
{

    field-type1 field-name1;
    field-type2 field-name2;
    field-type3 field-name3;

    ...
};
```

Variable declaration and Initialisation

- You must use keyword struct in the declaration

```
struct student s1;

struct car mycar;


struct student s1 = {“Nguyen Le”, 19, 8.0};

struct car mycar = {“Fiat”, "Punto", 2004};
```

Structure declaration with typedef

```
typedef struct student {
        char name[20];
        int age;
        float grade;
} student_t;

typedef struct car {
            char* make;
             char* model;
            int   year;
} car_t;
```

Now the program has a new types - **student_t and car_t**

Accessing Members of a Structure

- Use a dot between the structure name and the field name .

```
car_t mycar;

mycar.year = 2004;


student_t excellentp;

excellentp.age = 18;

excellentp.grade = 7.8;
```

Exercise 15.4

- Modify 15.2 to create a struct of students

```
typedef struct student {
        char name[30];
        char studentId[8];
        float grade;
} student_t;
```

- Similar to exercise 15.2, output the student details but with grades in descending order

Exercise 15.4.1

- Modify the student struct

```
typedef struct student {
        char name[30];
        char studentId[8];
        float course1Grade;
        float course2Grade;
        float course3Grade;
        char averageGrade; //A+,A,B+,B,
} student_t;
```

- Output the student details with average grades in descending order
  - Note: Two students with same average grade (A+,A,B+…) are considered equal grade