VG101 FA2021 RC #2

Foundations of C

Type Declaration

1. C is a statically-typed language, so any variable should be declared explicitly with a type.
2. Syntax:

```
typename var1, var2, ...;
```

3. Type decoration (such as *, &, etc.) can only decorate the following variable
4. Initialization (such as =) can only assign value to the previous variable

```
int *var1, var2;    // var1 is a "pointer to int", var2 is a "int"
int var3, var4 = 5; // var3 is uninitialized, var4 is initialized as
5
```

5. But C is not a strongly-typed language, for example

```
double num = 'a';
printf("num = %f\n", num);
// num = 97.000000
```

Block and Variable Scope

1. In C/C++, { } denotes a block
2. Local variable declared in a block will be destroyed when the block ends.

```c
int glob = 10;

int main() {
    for (int i = 0;i < 10;++i) {
        if (i >= 5) {
            int j = i;
        }
        printf("j = %d\n", j); // Error: j is undeclared
    }
    printf("i = %d\n", i); // Error: i is undeclared
}
```

Control Statements

1. `if` statement

   Syntax:

   ```c
   if (condition) {
       statement;
   }
   ```

2. `switch` statement

   Syntax:

   ```c
   switch (var) {
       case a:
           statement;
           break;
       case b:
           statement;
           break;
       default:
           statement;
   }
   ```

   `break` is necessary, otherwise after running "case a" it will automatically run "case b".

   Notice that statement is not contained in a block, so you are not allowed to declare a variable here.

Loop

1. `for` loop Syntax:

```
for (init; test; step) {
    statement;
}

// Any for loop can be equivalently converted into a while loop, and
vice versa.

init;
while (test) {
    statement;
    step;
}
```

From `lab3_Tutorial.c`:

```
int x = 0;
for (int i = 100; i > 1; i /= 2) {
    ++x;
}
// What is x now?
// i = 100 50 25 12 6 3 1(X)
```

2. `while` loop

Syntax:

```
while (condition) {
    statement;
}
```

`do...while` loop:

```
// Execute the statement at least once, and then test the condition
do {
   statement;
} while (condition);

// Generally equals to
statement;
while (condition) {
    statement;
}
```

3. How to convert the above `for` loop into a `while` loop?

```
int x = 0, i = 100;
while (i > 1) {
    ++x;
    i /= 2;
}
i = 2;

// What is the small difference?
```

Function

1. General syntax

```
return_type fun(type arg1, type arg2, ...) {
    statements;
    return some_value;
}

// Use return to end the function earlier

int strfind(char* str, char ch) {
    // If str contains ch, return the first index of ch
    // Otherwise return -1
    for (int i = 0; i < strlen(str); ++i) {
        if (str[i] == ch) {
            return i; // no need to check following characters
        }
    }
    return -1;
}
```

if there is no return value,

```c
void fun(type arg1, type arg2, ...) {
    statements;
}

// Use return to end the function earlier

void printPositiveNum(int num) {
    if (num <= 0) {
        printf("Non-positive number\n");
        return;
    }
    printf("num = %d\n", num);
}
```

2. Separation of declarations and definitions

   Functions should be declared before they're used. You may use

   ```c
   return_value fun(type arg1, type arg2, ...);
   ```

   to declare a dummy function first, and then write implementations anywhere that is linked to the program.

Note: The `int main()` function is essential for any C program!

Basic Unix-Style Commands

Widely-used command line interfaces

- Windows:
  - Press "Win + R", type "powershell" or "cmd"
  - Optional: Download "Windows Terminal" from Microsoft Store
- macOS:
  - Open Terminal in the Application (Zsh by default)
  - Optional: Download "iTerm2" from https://iterm2.com/

Useful Commands in VG101

1. `pwd`: Print name of current/working directory
2. `cd [dirName]` : Change directory

```
cd ~  # change to personal directory, /home/${username} on
macOS/Linux
cd /  # change to the root directory
cd .. # change to the upper directory of the current directory
cd -  # change to the previous directory
cd JI/VG101/RC/demo
```

3. `ls [OPTION]... [FILE]...`: List directory contents

```
ls ~   # list files and subdirectories under personal directory
ls -a  # list all files and subdirectories (including hidden ones)
under current directory
ls -l  # list files an subdirecotories using a long listing format
ls *.c # list files ended with ".c"
```

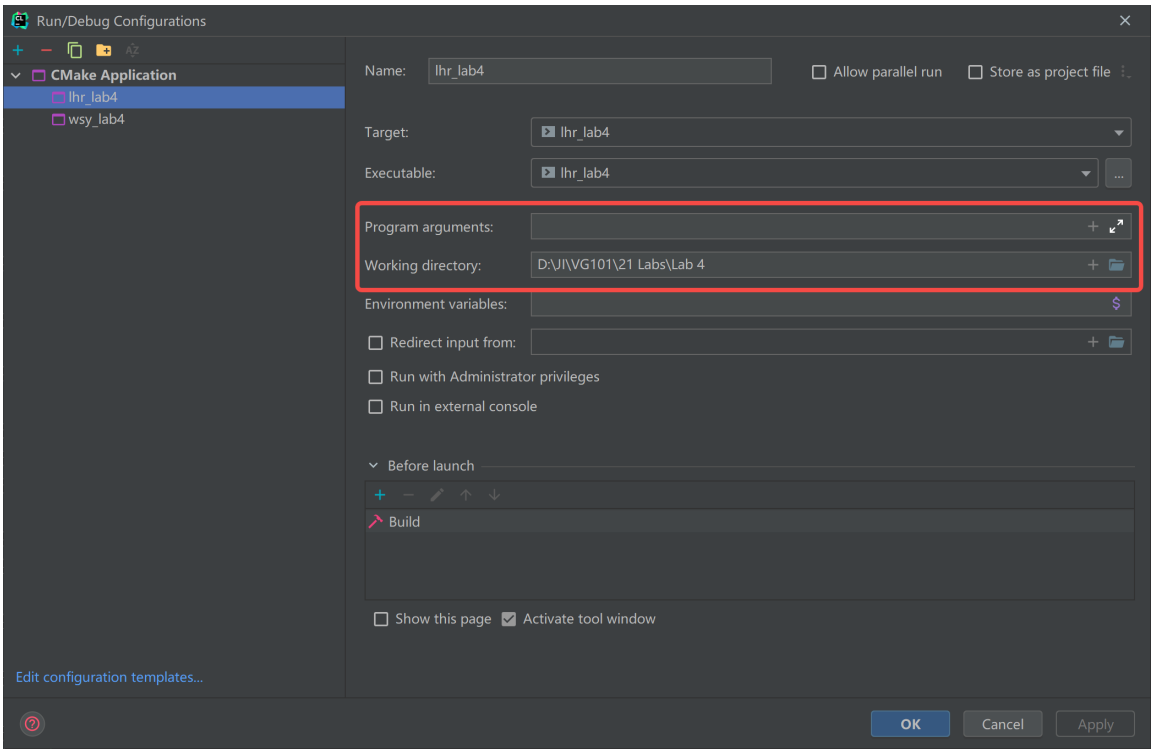
4. `cat [FILE]...`: Print the content of the file

```
cat lab3_Tutorial.c
cat CMakeLists.txt
```

5. `gcc`: GNU project C compiler

CLion/VSCode is powerful and convenient, why do we need to learn command line?

- Basic mechanism:
  - IDE like CLion just combines the works done by several commands into one button. For example when you click "Reload CMake Project", it actually does,
  - ```
    mkdir cmake-build-debug && cd cmake-build-debug
    cmake -DCMAKE_C_COMPILER=gcc ..
    ```
  - And when you click "Run", it does,
  - ```
    cd cmake-build-debug
    cmake --build . && ./${selected executable}
    ```
- More Flexible and Faster:
  - If your program accepts program arguments, and you want to test your program with different program arguments,
  - ```
    # In command line
    gcc -o OUTPUT INPUT.c
    ./OUTPUT argument1 argument2 ...
    ./OUTPUT argument3 argument4 ...
    # ...
    ```
    ```

- In CLion:
- 

- More powerful (After you master the command line)
  - A shell script helps to grade your mid1
  - ```
    filelist=`ls | grep -v Auto`
    for file in $filelist
    do
        if [ ! -f ${file}/Problem1.m ]; then
            echo $file | sed -rn 's/([^_]*)_.*([0-9]{12}).*/\1
    \2/p' # Print Name and SID
            cd $file
            my_path=`find . -name Problem1.m | awk -F '/' '{print
    $2}'`
            cp ${my_path}/Prob*.m .
            cd ..
        fi
    done
    ```

C Compilation Process & How to Compile a C Program Using GCC

Intermediate Compilation Process

```
gcc -E INPUT -o OUTPUT # Pre-process
gcc -S INPUT -o OUTPUT # Compiling
gcc -c INPUT -o OUTPUT # Assembling
gcc INPUT -o OUTPUT    # Linking / Whole Process
```
```

Compile C Program via Command Line

1. Compile ONE C code without extra libraries (except std libraries).

```
gcc INPUT -o OUTPUT
```

2. Compile ONE C code with **MATH library** (`math.h`)

```
gcc INPUT -o OUTPUT -lm
```

3. Compile **SEVERAL** C codes and link them together

```
gcc INPUT1 INPUT2 -o OUTPUT
```

4. Compile arguments we use in VG101

```
gcc -O2 -Wall -Wextra -Werror -pedantic -Wno-unused-result -std=c11
-o OUTPUT INPUTS -lm
```

Compile C Program via CMake in CLion

```
cmake_minimum_required(VERSION 2.7)
project(vg101_lab3) # Project Name

set(CMAKE_C_STANDARD 11) # C standard
set(CMAKE_C_FLAGS "-Wall -Wextra -Wconversion -Werror -pedantic -Wno-
unused-result") # compile flags

add_executable(lab3_Tutorial lab3_Tutorial.c) # Executable file for
Tutorial
add_executable(lab3_Mathsh lab3_Mathsh.c)     # Executable file for
Mathsh
add_executable(test test.c)                   # Executable file for
test.c
```

Lab 3 - Mathsh

```c
int a = 10, b = 4;
int x2 = (++b) || (a -= 10); // short circuit
// x2 = , a = , b =

int x3 = (a -= 10) || (b++); // short circuit
// x3 = , a = , b =

// About scanf
int num;
scanf("%d", &num);    // normal way
scanf("%d ", &num);   // add a space
scanf("%d\n", &num); // add a newline (\n)
```

The format string consists of a sequence of directives which describe how to process the sequence of input characters. If processing of a directive fails, no further input is read, and scanf() returns. A "failure" can be either of the following: input failure, meaning that input characters were unavailable, or matching failure, meaning that the input was inappropriate (see below).

A directive is one of the following:

- A sequence of white-space characters (space, tab, newline, etc.; see isspace(3)). This directive matches any amount of white space, including none, in the input.
- An ordinary character (i.e., one other than white space or '%'). This character must exactly match the next character of input.
- A conversion specification, which commences with a '%' (percent) character. A sequence of characters from the input is converted according to this specification, and the result is placed in the corresponding pointer argument. If the next item of input does not match the conversion specification, the conversion fails—this is a matching failure.

Lab 4 - MATLAB Interpreter

- Much harder than Lab 3
- My personal completion time and code length:
  - Lab 3: 5 mins, 50 lines
  - Lab 4: 4.5 hours, 300+ lines
- Therefore, **start early!**
- Time Division: Set some milestones, for example

6. Rubrics

1. Be able to calculate an expression(2.1): 120 pts
2. Be able to assign a value or expression to a variable, and calculate an expression with variables(2.2): 40 pts
3. Be able to output a string without %f: 20 pts
4. Be able to output a string with %f and \n(2.3), and complete this lab successfully: 20 pts
5. (bonus) Finish the lab in one week: 5% bonus(10 pts).

- Code Division: How to divide it into several parts? (Personal Experience)
  - Read from file (*easy*)
  - Parse the command into Infix Expression (*hard and tricky*)

    - Convert Infix to Reverse Polish Expression (*moderate ~ hard*)
    - Calculate the result of Reverse Polish Expression (*moderate*)
    - Print result / Assign result to variables (*moderate*)
- Which aspect of knowledge should you master for each part?
    - File I/O (`fopen, fclose, fgets`)
    - Loops, C-style string (`char *`)
    - Loops, Stack, Array (`double arr[]`)
    - Loops, Stack, Array
    - Output format, Array
- Write some helper functions, since there are lots of repetitive work
- **More details in next week RC**

General Tips

- Screenshots are clearer than photos taken by mobile phones.
- Read the error message carefully, and then search it via search engines.
- The biggest advantage of programming: you can always try it by yourself!