COMP 2012 Object-Oriented Programming and Data Structures

Lab 1 Makefile and Separate Compilation

Image Source: Bailey Mariner. Accessed via
https://www.healthline.com/health/apple-body-shape

Assessment of Level of Health Risk

Introduction

In this lab, we are going to practice separate compilations using Makefile. A collection of C++ source and header files associated with various tests on the level of health risk is provided to you. In particular, you are required to come up with an appropriate Makefile to generate specific tests on health risk based on Waist-to-Hip Ratio (WTH), Body Mass Index (BMI), and Body Fat Percentage (BFP). To get prepared for the task, you are advised to read the Makefile lecture notes and a quick tutorial below.

1. What is Separate Compilation?

Suppose you have a program that consists of 3 cpp files: `file1.cpp`, `file2.cpp`, and `file3.cpp`.

With the g++ command, one can compile the program with the following command that produces an executable named `a.out`

```
g++ -o a.out file1.cpp file2.cpp file3.cpp
```

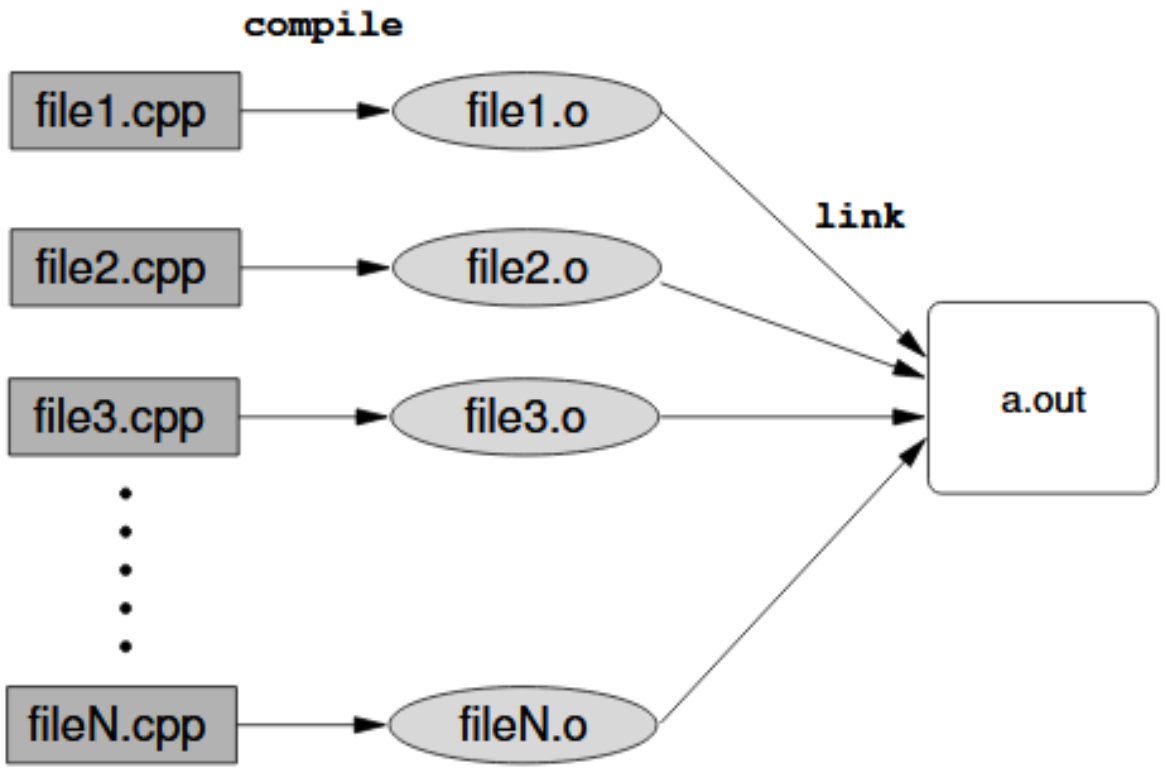
Any changes to 3 files (e.g., just editing a line in `file1.cpp`) will require the g++ compiler to recompile all 3 cpp files to generate the updated executable.

This is inefficient. You may think it does not matter when there are only 3 cpp files. However, for projects on a large scale, the amount of time it takes to compile the executable in this way is huge. To alleviate this problem, we can make use of separate compilations.

To perform separate compilations, we use the following commands instead.

```
g++ -c file1.cpp
g++ -c file2.cpp
g++ -c file3.cpp
g++ -o a.out file1.o file2.o file3.o
```

Each of the first 3 commands will individually compile `fileN.cpp` into `fileN.o` - an object file (which is not executable) containing the compiled code for the corresponding cpp file. The last command then simply links those object files together to generate the final executable a.out.



Now, say, if a line is edited in `file1.cpp`, we just need to execute:

```
g++ -c file1.cpp
g++ -o a.out file1.o file2.o file3.o
```

And an updated executable would be generated because the existing file2.o and file3.o can simply be reused.

2. Makefile

Makefile allows us to create a set of compilation rules for your programming project. It is very useful for separate compilation. A Makefile consists of a number of rules with the following syntax:

```
[Name of rule (target) #1]: [List of dependent files/targets, separated by a space]
[Press a tab character here][Command 1 to be executed]
[Press a tab character here][Command 2 to be executed]
...
[Press a tab character here][Command N to be executed]
[Empty lines....]
[Name of rule (target) #2]: [List of dependent files/targets, separated by a space]
[Press a tab character here][Command 1 to be executed]
[Press a tab character here][Command 2 to be executed]
...
[Press a tab character here][Command N to be executed]
[Empty lines....]
```

The name of a rule is also called its target. Each rule clearly defines the dependent files and procedures for the system to generate the target. When a rule is executed, Makefile will search whether all dependent files are present in the directory and up-to-date. If not, it will execute another rule with its target specified as the missing/outdated file to generate the dependent file. Then, after all dependent files are generated, the commands under the rule will be executed. Usually, the commands are a series of procedures to create the target file using the dependent files. We usually call the dependent files "dependencies".

We can ask Makefile to execute a rule for us by typing:

```
make[SPACEBAR][The target of the rule]
```

For example, if the Makefile contains a rule as the following:

```
main.exe: main.o helper.o
  g++ -o main.exe -std=c++11 main.o helper.o
```

By typing `make main.exe` in the terminal, we can execute the rule.

Makefile can consist of multiple rules. The first rule in a Makefile will be executed by default when you simply run the command `make` (without mentioning the target/rule name). In the given Makefile in this lab, it will be the rule "all". Note that, however, the first target is generally not necessarily called "all"; it can be anything you want.

We usually write a rule to clean up a folder by removing all the temporary files (e.g. executable and object files). In the given Makefile, to execute that, you will have to run the command `make clean`.

The way we write the rule with target `clean` is as follows. You should use different command depending on your Operating System:

```
clean:
  rm  -f  *.o *.exe *.d
# On Windows, use: del *.o *.exe *.d
# On Linux or MacOS, use: rm -f *.o *.exe *.d
```

You may observe that this rule removes the ".d" extension files. They are "dependency" files and we will talk about it later.

Note: the rule with target `clean` does not have any dependencies. We call that "Phony Targets". These "targets" are not files we want to generate, instead they serve as a name to execute the commands under the rule.

Note 2: it is possible to detect which Operating System is being used and pick the command automatically in the Makefile, but it is slightly complicated, so just write the command that works for the Operating System you are using (please edit the provided Makefile accordingly yourself).

2.1 Make and Remake

We will study the flow of make in this section. Suppose we have the following Makefile:

```
main.exe: main.o helper.o
  g++ -o main.exe -std=c++11 -Wall main.o helper.o

main.o: main.cpp
  g++ -std=c++11 -Wall -c main.cpp -o main.o

helper.o: helper.cpp
  g++ -std=c++11 -Wall -c helper.cpp -o helper.o
```

Assume we have 2 files: `main.cpp` and `helper.cpp` in our directory. By typing `make`, the Makefile will execute the first rule.

The following sequence of action is performed:

1. The Makefile check whether the first dependency `main.o` is in the directory.
2. Since `main.o` is not in the directory, the Makefile then finds another rule to make `main.o`.
3. The Makefile finds the rule with `main.o` as target, it finds the dependencies `main.cpp` in our directory.

4. All dependencies for the rule `main.o` is satisfied. The command `g++ -std=c++11 -Wall -c main.cpp -o main.o` is executed. `main.o` is created.
5. The Makefile check whether the second dependency `helper.o` is in the directory.
6. Similar to how `main.o` is created, `helper.o` is created by executing the rule `helper.o: helper.cpp`.
7. All dependencies for the rule `main.exe` is satisfied. The command `g++ -o main.exe -std=c++11 -Wall main.o helper.o` is executed. `main.exe` is created.
8. The Makefile finishes its job.

What if we change several lines in `helper.cpp`? What happens if we type `make` again? The following sequence of action is performed:

1. The Makefile check whether the first dependency `main.o` is in the directory.
2. Since `main.o` is in the directory, the Makefile have to further check whether `main.o` needs to be updated.
3. The Makefile checks all the dependencies of `main.o` in its corresponding rule, and finds out that no dependencies (i.e. `main.cpp`) are updated.
4. The Makefile cannot find any dependencies defined for `main.cpp`. Therefore, `main.o` is up-to-date.
5. The Makefile check whether the second dependency `helper.o` is in the directory.
6. Since `helper.o` is in the directory, the Makefile have to further check whether `helper.o` needs to be updated.
7. The Makefile checks all the dependencies of `helper.o` in its corresponding rule, and finds out that one of the dependencies `helper.cpp` is updated.
8. The Makefile executes the command under the rule `g++ -std=c++11 -Wall -c helper.cpp -o helper.o`, generate an up-to-date version of `helper.o`.
9. Since there is a change in the dependencies of `main.exe`, the Makefile executes the command under the rule. `g++ -o main.exe -std=c++11 -Wall main.o helper.o`
10. `main.exe` is up-to-date. The Makefile finishes its job.

From the above example, you can see how powerful Makefile is in generating the files we want. If you observe the remake process, the Makefile reuse `main.o` because `main.cpp` has not been edited. By defining the dependency relationship clearly, we can easily exploit separate compilation by Makefile.

An Analogy

If you struggle with the mechanisms of Makefile, here we will discuss one analogy to Makefile. You can imagine the Makefile is actually a cookbook. In a cookbook, there are a lot of recipes (rules) to teach you how to cook a dish (target). Usually, to make a dish, you need to collect all ingredients (dependencies) first, then follow the steps (commands) to cook a dish (produce the target).

When you are collecting ingredients, you will usually search in the fridge/inventory first (directory). If there is no such ingredient, you may have to make it using other recipes in the cookbook (find a rule to make it).

Note: there are subtle differences between cookbook and Makefile, e.g. you may use up some ingredients in cooking. For the Makefile case, as long as the file is in your directory, you can use it any number of times.

The following sections will introduce more advanced Makefile features that enable us to write Makefile easily.

2.2 Variable and Automatic Variable

When we write rules, we may repeat similar rules multiple times. If we use the previous example:

```
main.exe: main.o helper.o
  g++ -o main.exe -std=c++11 -Wall main.o helper.o

main.o: main.cpp
  g++ -std=c++11 -Wall -c main.cpp -o main.o

helper.o: helper.cpp
  g++ -std=c++11 -Wall -c helper.cpp -o helper.o
```

You can see that we repeat the g++ compiler flags `-std=c++11 -Wall` 3 times. Like in programming, Makefile provides the use of Variable. The syntax of defining a Variable in Makefile:

```
[identifier] = [the string it represents]
```

For example,

```
CPPFLAGS = -std=c++11 -Wall
```

We can access the variable by the following syntax:

```
$([identifier])
```

Using variables, we can simplify our Makefile like the following:

```
CPPFLAGS = -std=c++11 -Wall
OBJS = main.o helper.o

main.exe: $(OBJS)
  g++ -o main.exe $(CPPFLAGS) main.o helper.o

main.o: main.cpp
  g++ $(CPPFLAGS) -c main.cpp -o main.o

helper.o: helper.cpp
  g++ $(CPPFLAGS) -c helper.cpp -o helper.o
```

By using variables, now the Makefile is more readable.

To further simplify the command under each rule, we can use another feature - Automatic Variable. We will cover 3 Automatic Variables in this lab:

- `$@`: it represents the target of the rule.
- `$^`: it represents ALL the dependencies of the rule.
- `$<`: it represents the first dependency of the rule.

Using Automatic Variables, we can further simplify our Makefile to the following:

```
CPPFLAGS = -std=c++11 -Wall
OBJS = main.o helper.o

main.exe: $(OBJS)
  g++ -o $@ $(CPPFLAGS) $^

main.o: main.cpp
  g++ $(CPPFLAGS) -c $< -o $@

helper.o: helper.cpp
  g++ $(CPPFLAGS) -c $< -o $@
```

2.3 Pattern Rules

We are not done yet! You may observe the last 2 rules are very similar. They only differ by the names in the target and dependency.

```
main.o: main.cpp
  g++ $(CPPFLAGS) -c $< -o $@

helper.o: helper.cpp
  g++ $(CPPFLAGS) -c $< -o $@
```

We introduce another feature in Makefile - Pattern Rules. Pattern Rules are very similar to ordinary rules except the symbol `%` is used. `%` acts like a wildcard. For example, `%.o` will match any file name that ends with `.o`. `lab1_%.o` will match with any files starting with `lab1_` and ending with `.o`.

`%` can be used in target and dependencies. The part of string that matches with `%` in the target will be substituted to every `%` in the dependencies. For example, `%.o: %.c`. If `main.o` is matched as a target of this rule, then the `%` in the dependencies is substituted by `main`. This rule becomes `main.o: main.c`. In this case, we call `main` the stem.

Note: Makefile will always try to match a file to an ordinary rule first, if such a rule does not exist, it tries to match with pattern rules.

Note 2: If a file matches multiple pattern rules, it will either match with the first matched pattern rule or the pattern rule with the shortest stem depending on the Makefile version.

With the help of Pattern Rules, we can once again simplify the Makefile:

```
CPPFLAGS = -std=c++11 -Wall
OBJS = main.o helper.o

main.exe: $(OBJS)
  g++ -o $@ $(CPPFLAGS) $^

%.o: %.cpp
  g++ $(CPPFLAGS) -c $< -o $@
```

2.4 Advanced Makefile

We have studied the basics of Makefile. Now we will study a more advanced version. You can find this in the Makefile lecture note slide p.15. The source files can be downloaded from the course website. The Makefile is as follows:

```
# Definition of variables
SRCS    = bulb.cpp lamp.cpp lamp-test.cpp
OBJS    = $(SRCS:.cpp=.o)
DEPS    = $(OBJS:.o=.d)
EXE     = lamp-test
CXXFLAGS = -std=c++11

# Rules:
# target: dependencies
#   command used to create the target
$(EXE): $(OBJS)
  g++ $(CXXFLAGS) -o $@ $(OBJS)

# To include the .d dependency files
-include $(DEPS)

# -MMD -MP creates the .d dependency files
.cpp.o:; g++ $(CXXFLAGS) -MMD -MP -c $<

clean:; /bin/rm $(EXE) $(OBJS) $(DEPS)
```

There are several features in this Makefile that have not yet been introduced. We will talk about ".d" (dependency file) and header file dependencies first.

We can fully utilize Makefile only when we list out all dependencies. In our previous example, we list out the dependent ".cpp" file for each ".o" file. However, we do not list out the header files that are included in the ".cpp" file. When we edit ".h" file, although the content in ".cpp" file does not change, the content of the ".cpp" file after preprocessing will be different. This means the ".o" file should also depend on those ".h" files.

We may add the dependent header files in the rule for each object file manually, but that will be very painful. Luckily, we can generate these dependencies through the g++ flag `-MMD -MP`. When the ".cpp" file is compiled into ".o" file, a dependency file (.d) listing all dependencies of the ".o" file (including header files) is generated. An example is `lamp-test.d`:

```
lamp-test.o: lamp-test.cpp lamp.h bulb.h

lamp.h:

bulb.h:
```

We can include these ".d" files into our Makefile through `-include`. The syntax of `-include` is:

```
-include[SPACEBAR][FILE 1][SPACEBAR][FILE 2]...
```

We can use variables to simplify the `-include` command like the Makefile sample.

The content in the ".d" files are rules with target and dependencies, but without the command to compile the ".o" file. If we include the ".d" files into the Makefile, shouldn't we add back the commands manually? Makefile is very smart. It will try to generate the target by inferring the file type of the target and dependencies. If you want more details, you may refer to Implicit Rules in GNU make documentation.

You may wonder what is `$(SRCS:.cpp=.o)` used in Variable? This is another feature - Substitution References. It will take the value in variable `SRCS`, and replace instances of ".cpp" with ".o". In this way, we can create variables quickly.

3. Add the source code

We use VS Code for this semester. You may refer to our VS Code tutorial page for setup and usage instructions. You may also refer to our guide How to add "-std=c++11" to VS Code for help with compiling.

Download the source files. Unzip/extract the zip file and open the extracted folder. See Creating a project and using the terminal for the custom compilation command section of our VS Code tutorial on how to open the folder (you don't need to create a new folder because the folder is already there after the extraction). You may also find the instructions there useful in the latter part of this lab, where you will need to enter some commands in the terminal in VS Code.

4. Open the Makefile in VS Code

You should see a Makefile that is similar to the one we discussed:

```
CPPFLAGS = -std=c++11
SRCS = main.cpp risk_WTH.cpp
OBJS = main.o risk_WTH.o
DEPS = $(SRCS:.cpp=.d)

all: HealthCheck.exe

HealthCheck.exe: $(OBJS)
	g++ -o $@ $(CPPFLAGS) $^

%.o: %.cpp
	g++ -o $@ $(CPPFLAGS) -MMD -MP -c $<

-include $(DEPS)

clean:
	del *.o *.exe *.d
# On Windows, use: del *.o *.exe *.d
# On Linux or MacOS, use: rm -f *.o *.exe *.d

.PRECIOUS: $(OBJS) risk_WTH.o risk_BMI.o risk_BFP.o
```

Note 1: You must change `del *.o *.exe *.d` to `rm -f *.o *.exe *.d` under the "clean" rule if you are using Linux or MacOS.

Lab tasks

Overview

We finally finished all the setups and tutorials! Let's start exploring the collection of C++ source and header files associated with various tests on the level of health risk that has been made available (the download link is in the menu and previous section).

The program would ask for inputs on your gender and age as well as some additional information to give an estimate of your level of health risk ("LOW", "MODERATE", or "HIGH").

Note: While a lot of source files are provided, you don't need to read all of them to complete this lab. You will only need to work with `Makefile` and a few `.cpp/.h` files, with detailed instructions below.

Task 1: Compile and Generate the Program with Makefile

In the terminal (see the tutorial on how to open the terminal in VS Code), enter the `make` or `make all` command to compile the project using the Makefile. Observe how many files are compiled. You may also deduce the sequence of commands run by Makefile by doing a similar analysis to Section 2.3 in the Makefile tutorial above. Afterwards, you can enter the command `"./HealthCheck.exe"` in the terminal to run the generated program.

The following is a sample session, where the user inputs are shown in **bold** and underline:

```
Welcome to Health Check
Please enter F if you are a female, M if you are a male: F
Please enter your age: 32
Please measure your waist and enter the number in centimeters (cm): 86
Please measure your hip and enter the number in centimeters (cm): 100
*** Your WTH (Waist to Hip) Ratio is 0.86
*** Your body shape is FRUIT #3
*** Your level of health risk is HIGH
```

You might explore the program with your own data and see if you would agree with the level of health risk as estimated from the waist-to-hip ratio.

Task 2: Modify a Header File

The waist-to-hip ratio compares the difference in waist and hip circumference, which can help indicate fat distribution. A ratio greater than 0.85 in women and greater than 1.0 in men suggests greater fat stores in the stomach area. Those with a higher waist-to-hip ratio are at greater risk of chronic disease.

People have used fruit terms to describe body shapes for many years because this is an easy way to describe body types without using more scientific, formal terms.

- The "APPLE" body shape indicates that most of the fat is stored in the midsection and less fat is stored in the hips, buttocks, and thighs.
- The "PEAR" body shape indicates that more fat is stored in the hips, buttocks, and thighs than in the midsection.
- The "AVOCADO" body shape indicates something between "APPLE" and "PEAR"

What your Waist-to-Hip Ratio Means

WOMEN	HEALTH RISK	BODY SHAPE
0.80 or below	Low	Pear
0.81 to 0.85	Moderate	Avocado
0.85+	High	Apple

MEN	HEALTH RISK	BODY SHAPE
0.95 or below	Low	Pear
0.96 to 1.0	Moderate	Avocado
1.0+	High	Apple

In `risk_WTH.h`, line 17, change the constant variable

```
const char* BODY_SHAPE[3] = { "FRUIT #1", "FRUIT #2", "FRUIT #3" };
```

to more informative fruit items of "PEAR", "AVOCADO", and "APPLE" as suggested.

When first running `make`, you will get multiple ".d" files. By reading the ".d" files, you can observe that modifying `risk_WTH.h` will lead to modifying some specific object files. You can run `make` again in the terminal to verify which files are recompiled. Do you observe any difference compared to the first time running `make`?

Task 3: Generate Mulitple Files Using Makefile

You might have noticed that, in addition to the test to assess health risk based on the waist-to-hip ratio (WTH), we have also implemented two other tests based on body mass index (BMI) and body fat percentage (BFP), respectively.

You should do the following step-by-step in order to compile and generate three different implementations of the assessment of health risk:

1. In the Makefile, for the rule `all: HealthCheck.exe`, change it to

   ```
   all: HealthCheck_WTH.exe HealthCheck_BMI.exe HealthCheck_BFP.exe
   ```

2. In the Makefile, change the variables `SRCS` and `OBJS` to

   ```
   SRCS = main.cpp risk_WTH.cpp risk_BMI.cpp risk_BFP.cpp
   OBJS = main.o
   ```

   Notice that `OBJS` does not contain the `risk_*.o` files. Since each executable only needs one of these object files, which implemented the function `int get_risk_level(char, int)`, we put the remaining shared object files in the variable and can write the rule as:

   ```
   # Example for HealthCheck_WTH.exe
   HealthCheck_WTH.exe: $(OBJS) risk_WTH.o
   ```

Observe that there are three files (`risk_WTH.cpp, risk_BMI.cpp, risk_BFP.cpp`) in your directory which represents three different implementations of the assessment of health risk.

As the final step, you need to modify the Makefile so that it can create all three executables `HealthCheck_WTH.exe, HealthCheck_BMI.exe, HealthCheck_BFP.exe` with the corresponding `risk_WTH.cpp, risk_BMI.cpp, risk_BFP.cpp` files when typing `make all` in the terminal.

A naive way is to write 3 rules, 1 rule for each executable. In practice, this can be very inefficient due to the large number of files you might need to generate, so you are encouraged to try using Pattern Rule to generate all 3 files with a single rule definition.

Essentially, you have to modify the following components in the Makefile:

1. Executable generation rule: `HealthCheck.exe: $(OBJS)`.
2. Defined variables: `SRCS, OBJS`

Sample Session #1: Output of `HealthCheck_WTH.exe` with the sample test case

```
Welcome to Health Check
Please enter F if you are a female, M if you are a male: F
Please enter your age: 32
Please measure your waist and enter the number in centimeters (cm): 86
Please measure your hip and enter the number in centimeters (cm): 100
*** Your WTH (Waist to Hip) Ratio is 0.86
*** Your body shape is APPLE
*** Your level of health risk is HIGH
```

Sample Session #2: Output of `HealthCheck_BMI.exe` with the sample test case

```
Welcome to Health Check
Please enter F if you are a female, M if you are a male: F
Please enter your age: 32
Please measure your weight and enter the number in kilograms (kg): 88
Please measure your height and enter the number in meters (m): 1.48
*** Your BMI (Body Mass Index) is 40.18
*** Your category is OBESE
*** Your level of health risk is HIGH
```

Sample Session #3: Output of `HealthCheck_BFP.exe` with the sample test case

```
Welcome to Health Check
Please enter F if you are a female, M if you are a male: F
Please enter your age: 32
Please measure your weight and enter the number in kilograms (kg): 88
Please measure your height and enter the number in meters (m): 1.48
*** Your BFP (Body Fat Percentage) is 50.17%
*** Your category is OBESE
*** Your level of health risk is HIGH
```

Submission Deadline and Guidelines:

The lab assignment is due 10 minutes after the end of your lab session.
We will use the online grading system ZINC, to grade your lab work. Therefore, you are required to upload the following files **in one single zip file** to ZINC:

```
- Makefile
- risk_WTH.h
```

Please note that ZINC is only used as a submission platform this semester. It will not perform real-time auto-grading before the due date. You can submit your codes multiple times to ZINC before the deadline. Only the last submission will be graded.

Grading Scheme

Successful completion of the lab requires completing the following items:

1. Attend the lab
2. Submit your code to ZINC by the end of your lab session.
    - You get a point for this task if your submission (with the skeleton files) can run the `make all` command and three executable files (`HealthCheck_WTH.exe, HealthCheck_BMI.exe, HealthCheck_BFP.exe`) are generated.

- *Note: We will compare program output to ensure the executable files are properly compiled. Your programs must give EXACTLY THE SAME OUTPUT, as shown in the three sample sessions in order to receive full marks.*

3. Answer a question related to the lab to show that you really work out the solution yourself.
    - Note: Students will be randomly selected to answer a question about the lab. If you are selected, the answering point will be given only if you answer the question correctly. If you are not selected, you get the answering point automatically if you have finished the lab and submitted your code to ZINC for assessment.

The lab is worth 3 points, 1 point for each of the items above.

Extra Notes

In this lab exercise, you compile and generate 3 different programs to assess the level of health risk using Makefile. However, what if we want to allow the user to select the specific assessment test at the beginning of the program, or even present the results of all assessment tests to all users? You may think of using a switch statement. Indeed, it can achieve our target. However, you may have to write some lines of code like this:

```
switch(which_test){
  case 1: // Waist-to-Hip Ratio
      level = get_risk_level_WTH(gender, age);
      break;
  case 2: // Body Mass Index
      level = get_risk_level_BMI(gender, age);
      break;
```


meaning you need 3 functions. This makes the development process more painful because you may need to include many switch

Page maintained by

Namkiu Chan
Last Modified: 02/15/2023 08:47:43

Homepage

Course Homepage