Give C Another Chance

1 Selection Statements

1.1 If, if-else, and if-elseif-else statements

If, if-else, and if-elseif-else statements in C are similar in usage to the if statements in Python. They only differ in syntax.

1.1.1 If statements

```
if(boolean_expression){
    // code here
}
```

1.1.2 If-else statements

```
if(boolean_expression){
    // code here
}else{
    // code here
}
```

1.1.3 If-elseif-else statements

```
if(boolean_expression1){
    // code here
}else if(boolean_expression2){
    // code here
}else{
    // code here
}
```

1.2 Switch statements

Switch statements allows you to check whether a given value (from a variable or from an expression) is **equal** to any of the listed values, called cases, inside the statement.

Usage:

```
switch(expression){
    case constant_expression1:
        statements_if_satisfied;
        break;
    case constant_expression2:
        statements_if_satisfied;
        break;
    default:
        statements;
}
```

Note: The case keyword is required and must be followed by a constant value. In case no value in the given cases were satisfied, the switch statement will execute the code in the default section. If no default section is specified, it will just proceed to the statement after the switch.

2 Iterative Statements(Loops)

2.1 For Loop

```
for (initialization; condition; variable_update){
    // code here
}
```

2.2 While Loop

```
while(condition){
    // code here
}
```

2.3 Do-while Loop

```
do{
    // code here
}while(condition);
```

3 Continue and Break

3.1 Continue

Continue is used to ignore the next lines of codes in a loop immediately after this statement and continue on to the next iteration.

3.2 Break

Break is used in terminating the iteration of a loop or stopping the execution of commands in a switch case.

4 Makefile

Compiling and running several source codes is time-consuming especially when you have to type them over and over again every time you have an update to your files. The use of makefile will help us programmers to get away from command line compiling. Makefile uses the command make and the filename of a makefile is usually the makefile word itself.

Below is an example of the contents of a makefile:

```
1 files = program.c program.h
2
3  run: $(files)
4     gcc -o program program.c
5     ./program
6  clean:
7     rm -f $(files)
```

A makefile has two important elements:

4.1 Macros

Macros are just variables. It takes the form of `x = y`.
In the example above, one macro is declared and named `files`. It was used by enclosing the macro name with `$()`just like in lines 3 and 7.

What happens when you enclose a macro with $()?
The $() replaces the macro with its value. For instance, in line `7, rm -f \$(files)` will be expanded to `rm -f program.c program.h`.

Macros can be useful if you have a string of characters that occurs in the makefile multiple times. You only need to type the whole string of characters once (in the macro), and then use the macro all throughout the makefile. This will also be useful if, at some point, you need to change the filename of program.c. You only need to change the value in the macro.

4.2 Targets

Targets are the series of commands in the makefile. Each target has three main components: a) target name, b) target dependencies, and c) the commands associated with the target. It follows the format below:

```
target_name : [target_dependencies]
<tab> <command_1>

<tab> <command_n>
```

In the previous example, there are two targets: run and clean.
Target `clean` has no dependencies while target `run` will not execute unless the files specified (by the macro

`files`) exist in the directory. If a dependency is not found, the commands of the target will not be executed. The commands are the tabbed lines. Lines 4 and 5 is associated with target `run` while line 7 is associated with target `clean`.