Homework #6 - Preprocessor Directives

Name:

1 Intro

C++ programs are **compiled**. This means that they are run through another computer program (called a compiler) which turns them into **machine code**, which is just a series of instructions for the computer, encoded as patterns of "1" and "0". During that process, our C++ files get read by the compiler, sometimes more than once. The first step in creating our program is done by a part of the compiler called the **preprocessor**. The job of the preprocessor is to make changes to our code before it is turned into machine code. You have gotten used to starting your programs with:

```
#include <iostream>
```

But why do we do this? In C++, lines that start with "#" are special instructions for the preprocessor. The "#include <iostream>" tells the preprocessor to go find a C++ file called "iostream" and copy it into our file. We do this so that we can use "cin" and "cout". In fact, any time we want to include functions or tools from another C++ file we will use "#include". It is also possible to tell the preprocessor to replace certain words in our code with numbers, like this:

```
#include <iostream>
#define INF 1000

using namespace std;

int main() {
  int i = INF;
  cout << i << endl;
  return 0;
}
```

When we build and run this program, the INF will be replaced by the number 1000. We can also use this behavior to create something called a "macro". This lets us replace parts of our code with something that looks a bit like a function. For instance, like this (see next page):

```
#include <iostream>
#define min(a, b) a = a < b ? a : b

using namespace std;

int main() {
  int a, b;
  cin >> a;
  cin >> b;
  min(a,b);
  cout << a;
  return 0;
}
```

Here, the min(a,b) code will be **replaced** by the code from the #define, and will change "a" to be equal to the smaller number (either a or b).

A word of warning: The preprocessor is a powerful tool but it is very easy to use it wrong. Remember, the preprocessor is **not** smart. It simply looks for text that matches #define and then replaces it with whatever was in the #define. This can result in very unclear and strange behavior in your programs, and it can make them hard to debug, because the compiler will warn you about errors in your code which happened **after** the preprocessor runs...so the line numbers of the error messages may not match the line numbers in your file editor. Similarly it is easy to create a #define like this one which is **very** confusing for other programmers:

```
#include <iostream>
#define return return;}

using namespace std;

int main() {
  cout << "Hello world!" << endl;
  return
```

And this code **will** compile and run! Because "return" gets replaced with "return;}" which fills in the missing semicolon (;) and brace (}) for us. But it makes the program very confusing to read. So a word of warning: **if you must use #define, don't use it often, and only use it for very simple things**.

2 Now you try...

2.1 Making range(0,10) work in C++

Add one or more #define statements to this code so that it will run correctly. Do not change any of the code itself, just add #define statements.

```
#include <iostream>
using namespace std;
int main() {
  for range(0,10) {
    cout << i << endl;
  }
  return 0;
}
```

2.2 HackerRank Challenge

Solve the HackerRank C++ preprocessor challenge. You may find the hints below useful.

2.2.1 Hint: Deleting code with #define

You may need to add the following #define to ignore the "FUNCTION(a,b)" lines and instead write separate #define lines for the minimum() and maximum() functions:

```
#define FUNCTION(name,op) /* */
```

2.2.2 Hint: Doing if/else comparisons

You can do a really quick if/else statement in C++ like this:

```
(a == b) ? a:b
```

If a and b are equal, this code returns the number a. If a and b are not equal, this code returns the number b. Think about how you could use this with "<" and ">" to make **minimum** and **maximum** #define statements.

2.2.3 Hint: Adding quotes

It might also be useful for you to know that a #define like this...

```
#define makeString(thing) #thing
```

Will put quotes "" around whatever words you put into its parentheses (). You can use this to write a **toStr()** #define.