Problem Set - hellowho

Table of Contents

Getting Started

In this first problem set, we set up our programming environment on your computer as well as joining Piazza service. Also, we write the first program that accepts input from the console and process the input as requested.

From GitHub, get https://github.com/idebtor/nowic repository on your computer. Keep this repository as "read-only". Copy them into your own repository or development folder (e.g. ~/nowicx) in some place you easily access them. They should look like the following:

```
~/nowic/psets/pset1/pset1.pdf              # this file
~/nowic/psets/pset1/hellofunc.exe          # a solution to compare your work for Windows
~/nowic/psets/pset1/hellofunc              # a solution to compare your work for macOS
~/nowic/psets/pset1/names.txt              # a list of names used in Step 2.
```

Overview: Hello <Who>!

In this problem set, we want to learn

- g++ compilation and execution in a command line
- Using a new line escape character `\n` and `endl` manipulator
- Processing the command-line arguments passing through **main(int argc, char *argv[])**
- Handling i/o at console using **iostream**, **cout, cin, getline()**
- Using functions and its prototype
- Compiling and linking multiple source files.
- Piping

Step 1. Create a source file: hellowho.cpp

Write your source program, **~/nowic/pset1/hellowho.cpp** as shown below:

```
/*
* file: hellowho.cpp
* It prints "Hello World!" or "Hello" with a given name.
* The completed code should work as shown below. “>” is a prompt of the console.
*
* To run the program without a command line argument:
*   > ./hello
*   > Enter a name: John Lee
*   > Hello John Lee!
*   > Enter a name: Peter Kim
*   > Hello Peter Kim!
*   > Enter a name:<Enter>
*   > Hello World!
*   >
*
* To run the program with a command line argument:
*   > ./hello John "Dr. Lee" "Handong Global University" peter
*   > Hello John!
*   > Hello Dr. Lee!
*   > Hello Handong Global University!
*   > Hello peter!
*   > Hello World!
*   >
* To run the program through a pipe
* (names.txt contains a list of names as shown below:)
*   > cat names.txt | ./hello
*   > Enter a name: Hello john!
*   > Enter a name: Hello Dr. Lee!
*   > Enter a name: Hello Handong Global University!
*   > Enter a name: Hello Peter!
*   > Enter a name: Hello World!
*
*/

#include <iostream>
#include <string>
using namespace std;

int main() {
  // if necessary, use setvbuf() to prevent the output from buffered on console.
  // setvbuf(stdout, NULL, _IONBF, 0);

  cout << "Your code here" << endl;
  cout << "Hello World!\n";

  // if necessary, use system("pause") to prevent the terminal from disappearing
  // as soon as the program terminates as in Visual Studio sometimes.
  // system("pause");
  return EXIT_SUCCESS;
}
```

This program, **hellowho.cpp**, is eventually supposed to work as described in the comment section at the top of the file. The program works differently depending on whether or not there is a give command-line argument. If there is no command-line argument except the

executable itself (or hellowho.exe), it should ask the user to enter a name. Once the user enters a name, print "Hello" with the name. Otherwise finish the program with "Hello World!". If there are some arguments given, then assume that names are given. In the code, you use a loop statement to print them all and exit the program with "Hello World!".

Hint 1. Getting names from the user

You **must review** the lecture note titled "getinputs.md" available at:

nowic/pset1/**getinputs.md**

The key is to use `getline()` instead of `cin` as I told you during the lecture. If the user does not enter any name through the command-line, he should be able to enter a name repeatedly and interactively until he/she enters nothing or enter. Eventually the user enters <Enter>, quit the program with "Hello World!". The sample run is shown below:

Sample Run:

```
PS C:\GitHub\nowicx\psets\pset01hellowho> g++ hellowho.cpp -o hellowho
PS C:\GitHub\nowicx\psets\pset01hellowho> ./hellowho
Enter a name: john
Hello john!
Enter a name: Dr. Lee
Hello Dr. Lee!
Enter a name:
Hello World!
PS C:\GitHub\nowicx\psets\pset01hellowho>
```

Hint 2. Compiling and Linking

Compilation refers to the processing of source code files (.c, .cc, or .cpp) and the creation of an 'object' file. This step doesn't create anything the user can actually run. Instead, the compiler merely produces the machine language instructions that correspond to the source code file that was compiled. The following command line produces hellowho.o:

```
g++ -c hellowho.cpp
```

Linking refers to the creation of a single executable file from multiple object files. In this step, it is common that the linker will complain about undefined functions (commonly, main itself). During compilation, if the compiler could not find the definition for a particular function, it would just assume that the function was defined in another file.

The following command line links all object files necessary and produces an executable (hellowho.exe for PC or hellowho for macOS.)

```
g++ hellowho.o –o hellowho
```

You may do the compilation and linking in one command line if not many files are involved as shown below:

```
g++ hellowho.cpp –o hellowho
```

Step 2: Using the command line arguments

You may review Lecture Notes about command-line arguments available at: **nowic/pset1/argcargv.md.**

Make a copy of **hellowho.cpp** into **helloargs.cpp** using the following command.

```
$ cp hellowho.cpp helloargs.cpp
```

Now we want to use the command **line** to pass a list of names such that your program greets them individually. This part of the program needs to accept a command-line argument. Then, you need to declare **main()** in **helloargs.cpp** with:

```
int main(int argc, char *argv[])
```

The first argument **argc** has the number of arguments in the command line. For example, if a command line is set as shown below

```
./helloargs john "Dr. Lee" handong peter
```

Then **argc** and **argv** are set as shown below by the system automatically.

```
argc = 5
argv[0] = "C:/GitHub/nowicx/psets/pset1/helloargs"
argv[1] = "john"
argv[2] = "Dr. Lee"
argv[3] = "handong"
argv[4] = "peter"
```

Recall that **argv** is an "array" of strings. You can think of an array as row of gym lockers, inside each of which is some value (and maybe some socks). In this case, inside each such locker is a string. To open (i.e., "index into") the first locker, you use syntax like **argv[0],** since arrays are "zero-indexed." To open the next locker, you use syntax like **argv[1].** And so on. Of course, if there are **n** lockers, you'd better stop opening lockers once you get to **argv[n - 1],** since **argv[n]** doesn't exist! (That or it belongs to someone else, in which case you still shouldn't open it.) In other words, just as **argv** is an array of strings, so is a **string** an array of chars. And so you can use square brackets to access individual characters in strings just as you can individual strings in **argv.**

Notice that you need put a double quotation if an **argument** consists of more than one word. Eventually your code should be able to handle the command line arguments as shown below:

Sample Run: (Keep the same output when no command line argument is given.)

```
PS C:\GitHub\nowicx\psets\pset01hellowho> ./helloargs john "Mr. Lee" "Handong Global" peter
Hello john!
Hello Mr. Lee!
Hello Handong Global!
Hello peter!
Hello World!
PS C:\GitHub\nowicx\psets\pset01hellowho>
```

```
PS C:\GitHub\nowicx\psets\pset01hellowho> ./helloargs
Enter a name: Mr. Lee
Hello Mr. Lee!
Enter a name:
Hello World!
PS C:\GitHub\nowicx\psets\pset01hellowho>
```

Step 3: Building an executable using multiple files

In this step, we want to pass the names in the command line arguments to a function, **printfunc()** that is defined in a separate file called **printfunc.cpp** as shown below:

```
// file: printfunc.cpp
// C++ for C Coders & Data Structures
// Lecture note by idebtor@gmail.com
#include <iostream>

void printfunc(int n, char *names[]) {

    for (int i = 0; i < n; i++)
        std::cout << "Hello " << names[i] << "!" << std::endl;

}
```

Create **printfunc.cpp** as shown above.

Make a copy of **helloargs.cpp** into **hellofunc.cpp** using the following command.

```
$ cp helloargs.cpp hellofunc.cpp
```

- Modify **hellofunc.cpp** such that it produces the output like the previous steps. In order to use **printfunc()** in **hellofunc.cpp**, you must define its function prototype in **hellofunc.cpp**.
- Use **printfunc()** when user's input comes from command line argument.

Sample Run: (Keep the same output when no command line argument is given.)

```
PS C:\GitHub\nowicx\psets\pset01hellowho> ./hellofunc john "Mr. Lee" "Handong Global" peter
Hello john!
Hello Mr. Lee!
Hello Handong Global!
Hello peter!
Hello World!
PS C:\GitHub\nowicx\psets\pset01hellowho>
```

```
PS C:\GitHub\nowicx\psets\pset01hellowho> g++ hellofunc.cpp printfunc.cpp -o hellofunc
PS C:\GitHub\nowicx\psets\pset01hellowho> ./hellofunc
Enter a name: Peter Kim
Hello Peter Kim!
Enter a name:
Hello World!
PS C:\GitHub\nowicx\psets\pset01hellowho>
```

Hint 3. Compiling and Linking multiple files

As our programs get more complicated, we want to store the source code into separate files. For example, let's suppose we have `show()` defined in `show.cpp` and it is invoked in `show()` as shown below:

```
// file: show.cpp
#include <iostream>
void show(int a) {
    std::cout << a << std::endl;
}
```

To allow programs to be written in logical parts, C++ supports **separate compilation**. It lets us split our program into several files, each of which can be compiled independently. Since

`show.cpp` does not have `main()`, **the compiler cannot generate an executable**. But it can generates an **object** file with `.o` extension instead of `.exe`.

```
$ g++ -c show.cpp     # compilation: produces show.o
```

To generate an executable with the following file, `hello.cpp`, we must tell the compiler where to find all of the code we use.

```
#include <iostream>
void show(int a);   // lets the compiler know it from external sources
                    // this is called a function prototype
                    // the header file contains a list of function prototypes
int main() {
    show(123);
}
```

We might compile these files as follows.

```
$ g++ hello.cpp show.cpp -o hello      # compile & link: produces hello.exe
$ ./hello                              # runs hello.exe
```

Now we can run the `hello` program.

If we have changed only one of source files, we want to recompile only the file that actually changed. We should be able to have a way to separately compile each file. This process usually yields a file with the .o or .obj file extension, indicating that the file contains **object code**.

The compiler lets us **link** object files together to form an executable. On the system (Windows & gcc) we use, we would separately compile our programs as follows:

```
$ g++ -c show.cpp                  # compilation: produces show.o
$ g++ -c hello.cpp                 # compilation: produces hello.o
$ g++ show.cpp hello.cpp           # linking: produces a.exe
$ g++ show.cpp hello.cpp -o hello  # linking: produces hello.exe
```

More GCC Compiler Options

There are some flags available for compiler options:

```
$ g++ -std=c++11 -Wall file1.cpp file2.cpp -o prog   # generates prog.exe
```

- -o : specifies the output executable filename.
- -std=c++11 : to explicitly specify the C++ standard, c++11, c++14, c++17, c++2a
- -Wall : enables most warning messages.

A Good Guideline for GCC and Make :
[Compiling, Linking and Building C/C++ Applications]

Step 4: Using 'Pipe'

This part is to introduce you to the concepts of **'pipe'**. Without changing anything in your program, hopefully, it is supposed to run smoothly. It is supposed to get inputs (or names) from the output of the pipe process through redirection. You may use **cat** instead of **type** in PC/Windows. If this part does not work, you must fix your code to work as shown below:

Sample Run:

```
PS C:\GitHub\nowicx\psets\pset01hellowho> cat names.txt
john
Dr. Lee
Handong Global University
Peter
PS C:\GitHub\nowicx\psets\pset01hellowho>
```

```
PS C:\GitHub\nowicx\psets\pset01hellowho> cat names.txt | ./hellofunc
Enter a name: Hello john!
Enter a name: Hello Dr. Lee!
Enter a name: Hello Handong Global University!
Enter a name: Hello Peter!
Enter a name: Hello World!
PS C:\GitHub\nowicx\psets\pset01hellowho>
```

Submitting your solution

- Include the following line at the top of your every source file with your name signed.
  On my honour, I pledge that I have neither received nor provided improper assistance in the completion of this assignment.
  Signed: _____ Student Number: _____
- Make sure your code **compiles** and **runs** right before you submit it. Every semester, we get dozens of submissions that don't even compile. Don't make "a tiny last-minute change" and assume your code still compiles. You will not receive sympathy for code that "almost" works.
- If you only manage to work out the Project partially before the deadline, you still need to turn it in. However, don't turn it in if it does not compile and run.
- Place your source files in the folder you and I are sharing.
- After submitting, if you realize one of your programs is flawed, you may fix it and submit again as long as it is **before the deadline**. You will have to resubmit any related files together, even if you only change one. You may submit as often as you like. **Only the last version** you submit before the deadline will be graded.

Files to submit

- Submit your source files on time in the Piazza **pset1 folder**.
  - hellowho.cpp, helloargs.cpp, hellofunc.cpp
- Follow the TA's guideline when you turn in your file(s) since students from two sections are using the same file folder. Otherwise, there will be a penalty.
  Remember that your file submitted is kept with the time stamped.

Due and Grade

- Due: 11:55 pm