ECE 264: Advanced C Programming

Programming Assignment 1, Due 6/17/19

Welcome to ECE 264.

You will find this course challenging and rewarding. You will learn many things. This course will cover a wide variety of programming topics; one of the most important things you will learn is how to use tools that will make writing programs easier. This first assignment will help you learn some of those tools.

Before you learn any tool, you need to understand the purposes of these tools. These tools can help you understand your programs and detect potential problems. These tools are the software equivalents of oscilloscopes, telescope, ultrasounds, x-ray, MRI...

Even though learning these tools takes time, the benefits of these tools can save you much more time. Some students skip learning these tools hoping to save time, but they wind up losing much more time debugging their programs.

1 Learning Goals

You will learn to:

- Manage source code using the git version control tools
- Detect likely mistakes using gcc warnings
- Use conditional compilation to turn on and off sections of code
- Write a program that has multiple files
- Build a program and run multiple tests using make
- Use gcov to find untested code
- Use gprof to identify performance bottlenecks

2 Background

2.1 Version Control

Have you ever spent hours writing code (or a term paper) only to realize, when your computer crashes, that you haven't saved your progress? Have you ever spent an hour making a change to a program only to realize that all your changes were wrong and you had to go back to the way the code was? Version control helps solve these problems.

Every non-trivial program is written in stages. To write a good program, you need to divide it into stages and finish one stage at a time. After you finish one stage, commit a version so that you have a record. Version control provides many advantages. One of them is a simple way to back up your code. If, for any reason, you want to go back to a previous version, it is very easy. Version control does many more things than backing up your code.

Please understand that you must commit changes often if you want to use version control. If you do not commit, version control cannot help you.

This class uses git for version control. Git is a distributed version control system. That means there are two repositories: local and remote. When you commit changes, only the local repository is changed. This makes commits fast and independent of network connections. If your computer is damaged, you still lose the local repository.

To change the remote repository, you need to push the changes. If your computer is damaged, you can retrieve the code from the remote repository.

Please read the guide at github about how to use version control.

https://guides.github.com/

Please remember that you must commit and push often to take advantage of version control.

Version control is required in this class, so the instructors will not accept any excuse like "I accidentally deleted my code and please give me an extension." Neither will "my computer crashed" be accepted as an excuse.

You must commit and push often to demonstrate your progress of the assignments. If there is any doubt about academic dishonesty, your commit history would be an important piece of evidence proving your innocence.

2.2 GCC warnings

Before you write any program, you must assume that the program will have many mistakes and develop strategies to prevent, detect, and remove the mistakes. As explained earlier, many tools can help you and you must learn these tools.

When you write a program, you create text files. These text files are not computer programs. To convert the text files to machine-readable format, you need to use a compiler. This class uses `gcc` for the compiler.

texttgcc takes one or several `.c` files as input. If you want to specify the output file's name add

```
-o name
```

If you do not specify the name, the default name is called `a.out`

Please be careful not to use the file you write as the output's name, for example

```
> gcc myprogram.c -o myprogram.c
```

This will erase `myprogram.c`.

If you read the manual (also called the "man page") of `gcc`, you can find many options. In this class, you should *always* use `gcc` in the following way.

```
> gcc -std=c99 -g -Wall -Wshadow -pedantic -Wvla -Werror
```

- `-std=c99` means using the C standard announced in 1999
- `-g` means enabling debugging
- `-Wall` means enable warning messages
- `-Wshadow` means detecting shadow variables
- `-pedantic` means restricting standard C
- `-Wvla` means detecting variable length array
- `-Werror` means treating warnings as errors

You should create an alias so that whenever you use `gcc`, all these options are automatically added. It is very important to use `gcc` this way to detect mistakes. This is your first "line of defense" writing good programs.

There were many cases when students forgot to use these options, failed to detect these problems early, and spent many hours debugging.

Here are some common warning messages:

- create a variable but never use it (most likely it is mistyped)
- create a function but never use it (most likely it is mistyped)
- read a variable before a value is assigned (C does not initialize
- variables. If a variable is not assigned, its value is garbage, not necessarily zero)
- write code that can never be reached
- assign a value to a wrong type
- create two variables that have overlapping scopes (called shadow variables)

Remember, create an alias for `gcc` to include these options. If you do not create an alias, it is very likely that you forget.

2.3 Conditional Compilation

In some cases, you want to turn on or off sections of code. For example, you may want to print messages showing the progress and states of your programs, such as

```
printf("The value of x is %d\n", x);
```

Such a message should not occur after you finish the programs. Deleting these messages can be tedious. Moreover, when you delete the messages, you may accidentally delete other lines.

Fortunately, there is an easy solution: enclose the message by conditions using `ifdef` and `endif`:

```
#ifdef MESSAGE
printf("The value of x is %d\n", x);
#endif
```

If you add `-DMESSAGE` after `gcc`, this line is included in the program. If you do not add `-DMESSAGE` after `gcc`, this line is excluded in the program. What does `-DMESSAGE` mean? `-D` means defining the `MESSAGE` symbol. If `MESSAGE` is defined, the `#ifdef` condition is true and that line is included.

You can define multiple symbols. For example, if you want to test two different solutions of the same function, you can do the following:

```
#ifdef SOLUTION1
void func(....) // solution 1
{
        ....
}
#endif

#ifdef SOLUTION2
void func(....) // solution 2
{
        ....
}
#endif
```

If you have `-DSOLUTION1` after `gcc`, the first solution is included. If you have `-DSOLUTION2` after `gcc`, the second solution is included.

If you have neither, neither solution is included. You will get an error message because `gcc` does not know how to handle the call of `func`.

If you have both, you will get another error message because `gcc` does not know which one to use.

2.4 Multiple Files

Every non-trivial system is composed of multiple components.

C uses two types of files: header (`.h`) and source (`.c`). Header files contain declarations of functions and types. Source files implement functions. Header files are "included" and source files are "linked". To link source files, simply put the list of source files after `gcc`.

2.5 make

By now, you may feel that there is a lot of work to do before you write any code. If you feel that you need to type a lot of things after `gcc`, you are correct. Fortunately, many tools have been created to greatly simplify things.

All you need to do is five keystrokes: `make [Enter]`.

`make` is a Linux command. You need to write a file telling `make` what to do. This file, by convention, is called `Makefile`.

`Makefile` can also specify the sets of tests to run.

A Makefile may contain multiple sections. Each section has three parts:

```
target: dependence
[TAB] action
```

For example

```
testgen: testgen.c
	gcc testgen.c -o testgen
```

This means the program `testgen` depends on `testgen.c`. If `testgen.c` is changed, `gcc` will be called to compile `testgen.c` and generate `testgen`. Note that it is important that there be a TAB character before the action (not spaces).

By convention, an executable file in Linux has no extension (no `.c`, no `.h`, no `.exe`).

You can define symbols in Makefile. For example,

```
CFLAGS = -std=c99 -g -Wall -Wshadow -pedantic -Wvla -Werror
```

defines the flags mentioned earlier.

Please carefully study the Makefile included in this assignment. Note, especially, the first few lines of the Makefile. These define macros that can be used elsewhere to avoid typing the same thing over and over again. We use macros called `CFLAGS` and `GCC` to make sure that whenever we invoke `gcc`, we use the command line flags specified above.

2.6 Test Coverage

You would not write code for no purpose. You would think every line of code does something meaningful. However, sometimes, a program's conditions make certain portions of code unreachable. This is called "dead code". Here are two examples of dead code.

```
if ((x > 1) && (x < 0))
{
  // cannot reach here
}

int func(...)
{
    return ...;
    // whatever after return cannot be reached
}
```

Sometimes, the program's execution flow is not expected. Consider the following example.

```
if (x > 0)
{
    // do something
}
else
{
    // do something else
}
```

You may expect that `x` is greater than zero and the program should "do something". However, `x` is actually not greater than zero and the program does "something else". You can use many methods to detect such an unexpected program execution flow. One of the methods is called test coverage. If `x` is not greater than zero, then the code within the braces is not tested.

To use `gcov`, add

```
-fprofile-arcs -ftest-coverage
```

after `gcc`.

Then, execute the program as normal. To see the coverage of a particular source file type

```
> gcov filename.c
```

If any line is marked `#####`, this line is not tested.

2.7 Performance Profile

Computers were invented to accelerate computation. Thus, designing efficient algorithms is important. Finding performance bottlenecks (also called performance bugs) is not always easy. A tool called `gprof` samples where a program spends time and helps identify performance bottleneck.

To use `gprof`, add

```
-pg
```

after `gcc`

Execute the program and use

```
> gprof
```

to produce the profiling output.

2.8 Sorting

Sorting means arranging data in ascending or descending orders. Sorting is widely used in many applications. For exampling, Blackboard can sort students by their last names. Many sorting algorithms have been invented. Sorting algorithms can be evaluated using different metrics. The most commonly used metrics are the number of comparisons and the number of data movements.

This assignment asks you to implement the selection sort and to compare the selection sort and the quick sort. Quick sort takes advantage of transitivity: If a $>$ b and b $>$ c, then a $>$ c for sure. Thus, quick sort avoids comparing two numbers (a and c) if their order is already known.

You can use `gprof` to see the difference of quick sort and selection sort. It is likely that quick sort is much faster than selection sort. If the original data is already sorted, then quick sort is not faster because it cannot use transitivity. The algorithm for quick sort will be explained later this semester.

2.9 How to Write High-Quality Software

If you ask the students that have taken ECE 264, many will tell you that ECE 264 "transformed" them from casual programmers to serious software developers. To put it another way, in ECE 264, you learn how to develop serious software that may be used by others.

Do you notice the selection of words? Programmer and Software Developer? Are they different? Yes, they are different.

Developing software is much more than writing code (i.e., programming). As a rough analogy, developing software is like building a house and writing code is like laying bricks. Building a house is much more than laying bricks. Before laying down bricks, you need to design the house. How many floors? Is there a basement? How many rooms? Is there a garage? How would electrical wires, air flow, and water pipes be connected?

Imagine laying bricks for a house before you have a design. It would be a total disaster.

Many students, however, want to start writing code before they have designed the software. The results would not be desirable.

Many students mistakenly believe that they should spend most time debugging. They want to write code quickly and then debug. This is completely wrong. You should spend most time designing, not debugging.

When you design software, you need to think about how to test. If you believe that the software is correct, you would not fully test it.

Many students think developing software is about creativity and they cannot be restricted by rules. This is wrong. Every creative person must fully understand their tools and the boundaries of these tools. A musician must know many melodies. A painter must know how colors interact. A gymnast needs to follow the law of gravity. An interior designer must understand sunlight and seasons. A software developer needs to understand the process of creating software.

This assignment teaches you many important steps in developing software. In ECE 264, you must develop a strategy of planning before writing code. If you rush to write code, it is very likely that your programs will be full of mistakes (bugs) and you will spend much more time debugging.

Remember: spend time on design, not on debugging.

Another common mistake is to start coding without reading and understanding the entire assignment's requirements.

Every software developer needs to know debugging process. The best strategy, however, is to prevent bugs through careful design. Then, it is important to detect bugs automatically. In PA01, a function checks whether an array has been correctly sorted. Such checking code should be inserted into strategic locations while developing a large program. The checking code can detect problems as soon as they arise.

2.10 Limit Core

Coredump may be generated when your program terminates abnormally. Coredump can be a very large file and use up your disk quota. It is unlikely that you would debug using coredump. Thus, it is recommended that you limit the size of coredump to zero.

3 Instructions

You need to thoroughly understand every detail in this assignment. This assignment introduces many tools. You need to understand these tools because they are helpful in writing better programs.

3.1 Set up your Github account

Create a Github account (if you do not already have one). This is the account you should use to create and submit all of your assignments this semester.

Fill the Google form sent earlier and inform the TA and Instructors with your GitHub username by Thursday, 6/13.

Your github account should be your name (or something very close to your name, if someone else has taken the account of your name). Do not use any funny account name.

Please understand why most companies assign fistname.lastname for their employees' accounts. Nobody has time to remember the relationships between account names and real names. The easiest solution is to use real names for account names.

The teaching staff reserves the right to reject your assignments if your github account does not reflect your name.

1. Create a git repository for the assignment.

   Log in to your Github account. Then Then log in to ECE 264's Blackboard account, and find the announcement for PA01 and click the link. This will create a repository on Github for the assignment (you will follow a similar procedure for all future assignments). Make sure that the repository is called 'ECE264/PA01-<your username here>'.

2. Clone the repository to develop your assignment

   Cloning a repository creates a local copy. Change your directory to whichever directory you want to create your local copy in, and type:

   ```
   > git clone https://github.com/ECE264/PA01-<your username here> PA01
   ```

   This will create a subdirectory called PA01, where you will work on your code.

   In this command: `git clone` copies a repository. `https://github.com/ECE264/PA01-<your username here>` tells `git` where the server (remote copy) of your code is. `PA01` tells git to place the code in a local directory named `PA01`

   If you change to directory `PA01` and list the contents, you should see the files you will need for this assignment:

```
> cd PA01
> ls
```

And you should see all of the files, including the file you will need to edit, `selectsort.c`

3. As you develop your code, you can commit a *local* version of your changes (just to make sure that you can back up if you break something) by typing:

   ```
   > git add <file name that you want to commit>
   > git commit -m ``<describe your changes>''
   ```

   `git add <filename>` tells git to "stage" a file for committing. Staging files is useful if you want to make changes to several files at once and treat them as one logical change to your code. You need to call git add every time you want to commit a file that you have changed.

   `git commit` tells git to commit a new version of your code including all the changes you staged with `git add`. Note that until you execute `git commit`, none of your changes will have a version associated with them. You can commit the changes many times. It is a good habit committing often. It is very reasonable if you commit every ten minutes (or more often).

   Do not type `git add *` because you will likely add unnecessary files to the repository. When your repository has many unnecessary files, committing becomes slower. If the unnecessary files are large (such as executables or core files), committing can take several minutes and your assignments may be considered late.

4. To copy your changes back to Github (to make sure they are saved if your computer crashes, or if you want to continue developing your code from another machine), type

   ```
   > git push
   ```

   If you do not push, the teaching staff cannot see your solutions.

3.2 Understand the Tools

This assignment introduces many tools (`git`, `gcc`, `make`, `gcov`, `gprof`). You must be familiar with these tools.

3.3 Write the Selection Sort Function

Among all files given to you, you can modify only

`selectsort.c`

Modifying any other file may make your code incompatible with the grading program. You must type

```
> make inspect
```

and check

`selectsort.c.gcov`

If any line is marked `#####`, this line is not tested. You will lose 5% points for each line that is marked `#####`.

You must implement selection sort and not any other sorting algorithm. The teaching staff will use multiple methods to check whether you implement selection sort. If you implement any other sorting algorithm, you will lose as much as 50% points of this assignment.

You are strictly forbidden to implement bubble sort. Many students mysteriously love bubble sort. Bubble sort is inefficient because it moves data excessively. There is no known situation where bubble sort is better than all other sorting algorithms. There are many situations where bubble sort is worse than many other sorting algorithms. If you do not know bubble sort, you have missed nothing. You already know bubble sort, it is time to forget it.

3.4 Grading

If your program has any compilation error or warning (remember to use `gcc -std=c99 -g -Wall -Wshadow -pedantic -Wvla -Werror`), you will receive zero in this assignment.

In absolutely no circumstance can the teaching staff modify your program for grading. You cannot say, "If you change this, my program works." If your program misses a semicolon and cannot compile, you will receive zero. Your score depends on what is submitted, nothing else.

This assignment has given you many test cases. It is possible (and likely) that additional test cases will be used for grading. If your program passes (i.e., sorts the data correctly) some test cases and fails the others, the score will be proportional to the number of passed test cases.

3.5 Submitting your code

You will use git's "tagging" functionality to submit assignments. Rather than using any submission system, you will use git to *tag* which version of the code you want to grade. To tag the latest version of the code, type:

```
> git tag -a <tagname> -m ‘‘<describe the tag>’’
```

This will attach a tag with name <tagname> to your latest commit. Once you have a version of your program that you want to submit, run the following commands:

```
> git tag -a submission -m "Submission for PA01"
> git push --tags
```

This will create a tag named "submission" and push it to the remote server. The grading system will check out *whichever* version you have tagged "submission" and grade that.

If you want to update your submission (and tell the grading system to ignore any previous submissions) type:

```
> git tag -a -f submission -m "Submission for PA01"
> git push --tags
```

This will overwrite any other tag named submission with one for the current commit.

Please be careful about the following rules:

1. For each assignment, you should tag only one version with "submission". It is your responsibility to tag the correct one. You CANNOT request regrading if the grading program retrieves the version that you do not want to submit.
2. After tagging a version "submission", any modifications you make to your program WILL NOT BE GRADED (unless you update the tag, as described above).
3. The grading program starts retrieving soon after the submission deadline of each assignment. If your repository has no version tagged "submission", it is considered that you are late.
4. The grading program checks every student's repository 120 hours after the submission deadline. If a version tagged "submission" is found, the grading program retrieves and grades that version.

5. The grading program uses only the version tagged “submission”. It does NOT choose the higher score before and after the submission deadline. If a later version has the “submission” tag, this later version will be graded with the late discount. Thus, you should tag a late version with “submission” only if you are confident that the new score, with the late discount, is higher.
6. The time of submission is the time when you push the code to the repository, not the time when the grading program retrieves your code. If you push the code after the deadline, it is late. Even though you push before the grading program starts retrieving your program, it is still considered late.
7. You should push at least fifteen minutes before the deadline. Give yourself some time to accommodate unexpected situations (such as slow networks).
8. You are encouraged to tag partially working programs for submission early. In case anything occurs (for example, your computer is broken), you may receive some points. Please remember to tag improved version as you make progress.
9. Do not send your code for grading. The only acceptable way for grading is to tag your repository.

3.6 DO NOT SEND YOUR CODE TO TEACHING STAFF

Under absolutely no circumstance will the teaching staff (instructors and teaching assistants) debug your programs without your presence. Such email is ALWAYS ignored.

If you need help, go to office hours, or post on Piazza.

3.7 EXCEPTIONAL SITUATIONS

If there is a campus-wide situation (such as severe weather or power outage), the instructors will extend submission deadlines.

If you encounter an exceptional situation (such as injury in a traffic accident), the instructors will grant extensions based on the recommendations of the proper authorities (such as medical doctors or police). You are responsible providing proper documents requesting the extensions.

The teaching staff will not respond to email marked “URGENT”. For all emergencies, please call 911.