University of Reading
Department of Computer Science

Julian Kunkel, Oliver Barnwell

Exercise Week 3
CS1PR / Autumn 2019
30 Marks Total
Due Date: 2019-10-21 09:00

---

1. Make sure to plan your time for the whole sheet carefully. The complete exercise should represent approximately five hours of independent study. Each mark attained should correspond to roughly 10 minutes of your time and includes the time to check the lecture notes. If you are struggling, then reach out for help at `http://cs-reading.slack.com`. Click here to join.

2. Remember to store the files in your CS-Git repository and use `https://hps.vi4io.org/cs1pr-submit` to upload and verify your submission.

3. Follow our code of honour: do not copy and plagiarise. Any contribution made by others must be referenced correctly. Further information was given in the first tutorial. Before you attempt to look up a solution on the Internet (or with colleagues), try for at least 1/3rd of the time to find a solution by yourself.

4. The table of contents contains the regular tasks and the alternative HARDER assignments marked in red. Only attempt the hard tasks if you feel confident or have extra time to spend.

5. Do not forget to check if there was marked groupwork as part of the tutorial. The marks of these will be added to those of this exercise sheet. You can still complete it with a colleague.

Contents

Task 1: Understanding Compiler/Syntax Errors (4 Marks)

Understanding the output of the compiler is important to be able to fix the code. We're now going to look at some errors that are likely to occur while you're compiling your C programs, especially as a beginner. Such errors may happen, for example, when copying existing programs from the slides, but they can also be programming mistakes. It is your task to repair the programs such that they compile and produce the intended output.

We will list some common scenarios in which these errors happen along with the error message that the **gcc** compiler will output. Typically, when the compiler outputs an error, it includes the filename and line number in which the error occurs. In some cases, an error generates multiple false following errors, hence it is important to concentrate on the first appearing errors.

As part of this task, we provide several programs with syntactical errors that prevent them to be compiled and, thus, they generate compiler errors. To compile the examples, we recommend to use:

```
$ gcc -Wall -Wextra -Werror -o <FILE.EXE> <INPUT.C>
```

After each example, download the provided broken code example and create a new file which fixes the error and compiles successfully, giving the expected output.

1. **Source code:** (Click on the icon to download the file!)

```c
#include <stdio.h>

int main() {
  printf("hello, world!\n")
  return 0;
}
```

Syntax Error (missing semicolon):

```
  error1.c: In function 'main':
  error1.c:5:3: error: expected ';' before 'return'
     return 0;
     ~~~~~~
```

Semicolons are used to terminate code lines in the C programming language. Without them the C parser will not know when to stop parsing individual lines of code and will throw the above error, telling you to correct the error.

Expected Output:

```
hello, world!
```

2. **Source code:**

```c
#include <stdio.h>

int main() {
  printf("hello, world);
  return 0;
}
```

Syntax Error (missing terminating quotation mark):

```
  error2.c: In function 'main':
error2.c:4:10: error: missing terminating " character [-Werror]
   printf("hello, world);
          ^
error2.c:4:10: error: missing terminating " character
```

```
   printf("hello, world);
          ^~~~~~~~~~~~~~~~
error2.c:5:3: error: expected expression before ‘return’
   return 0;
   ^~~~~~
error2.c:6:1: error: expected ‘;’ before ‘}’ token
 }
 ^
```

A common mistake when defining strings in the source code is to forget a terminating quotation mark. The parser expects the string to be closed. Otherwise, it has no way of telling when the programmer has finished with their string definition. You have to close everything that comes in pairs, including:

- `[...]` – square brackets, used when defining and accessing arrays
- `{...}` – curly braces, used to define code blocks for functions and control statements
- `‘.’` – single quotes, used to define character constants (single chars) in the code
- `"..."` – double quotes, used to define string literals (sequences of characters)

Expected Output:

```
hello, world
```

3. **Source code:**

```
#include <stdio.h>

int main() {
  int 1nvalid = 1;
  printf("Number one: %d\n", 1nvalid);
  return 0;
}
```

Syntax Error (caused by invalid variable name):

```
  error3.c: In function ‘main’:
  error3.c:4:7: error: invalid suffix "nvalid" on integer constant
   int 1nvalid = 1;
       ^~~~~~~
error3.c:4:7: error: expected identifier or ‘(’ before numeric constant
error3.c:5:30: error: invalid suffix "nvalid" on integer constant
   printf("Number one: %d\n", 1nvalid);
                              ^~~~~~~
```

This is an example of an **invalid** variable name in C. Variables must not begin with an integer constant. Examples of other invalid variable definitions are:

- `int my variable` – no spaces are allowed in variable names
- `int $variable` – the $ character is not permitted
- `int while` – keywords such as `while, if, continue` are not allowed as variable names.

As a general rule, make sure that your variable names are descriptive and provide maximum clarity as to their purpose in the code. See the literature section if you want to read more on this topic.

Expected Output:

```
Number one: 1
```

4. **Source code:**

```c
#include <stdio.h>

int main() {
  printf("this isn’t a
      multiline
      string!");
  return 0;
}
```

Syntax Error (caused by not using newline characters):

```
      error4.c: In function ‘main’:
error4.c:4:10: error: missing terminating " character [-Werror]
   printf("this isn’t a
          ^
error4.c:4:10: error: missing terminating " character
   printf("this isn’t a
          ~~~~~~~~~~~~~~
error4.c:5:7: error: ‘multiline’ undeclared (first use in this function); did you mean ‘getline’?
       multiline
       ~~~~~~~~~
       getline
error4.c:5:7: note: each undeclared identifier is reported only once for each function it appears in
error4.c:6:7: error: expected ‘)’ before ‘string’
       string!");
       ~~~~~~
error4.c:6:14: error: missing terminating " character [-Werror]
       string!");
              ^
error4.c:6:14: error: missing terminating " character
       string!");
              ~~~
error4.c:8:1: error: expected ‘;’ before ‘}’ token
 }
 ^
```

When you want to include strings with multiple lines, it might be tempting to try and define them as we have done in the related code example. Unfortunately, this is not going to work, as to the C parser, it simply looks like an unterminated string on line 2 followed by simply the word **multiline**, which is invalid syntax!

If you want to achieve this in your own programs, use the \n character to represent new lines (automatically parsed by the terminal and displayed as new lines).

Expected Output:

```
this is a multiline string!
```

5. **Source code:**

```c
#include <stdio.h>

int main() {
  printf('Do not use single quotes for string literals!');
  return 0;
}
```

Syntax Error (caused by using single quotes for a string literal):

```
    error5.c: In function ‘main’:
error5.c:4:10: error: character constant too long for its type [-Werror]
   printf(‘Do not use single quotes for string literals!’);
```

```
          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
error5.c:4:10: error: passing argument 1 of ‘printf’ makes pointer from integer without a cast [-Werror=int-conversion]
In file included from error5.c:1:0:
/usr/include/stdio.h:318:12: note: expected ‘const char * restrict’ but argument is of type ‘int’
 extern int printf (const char *__restrict __format, ...);
            ^~~~~~
error5.c:4:3: error: format not a string literal and no format arguments [-Werror=format-security]
   printf('Do not use single quotes for string literals!');
   ^~~~~~
```

Single quotes are reserved in C for use only when defining single characters in your code. If you want to write a string literal (containing more than one character), you must use double quotes.

Expected Output:

```
Do not use single quotes for string literals!
```

Further Reading

- Camel Case, a commonly used convention for multi-word variable names.

  `https://en.wikipedia.org/wiki/Camel_case#Computer_programming`

- A helpful article on good naming conventions when programming.

  `https://hackernoon.com/the-art-of-naming-variables-52f44de00aad`

Submission (directory: `3/syntax-error-easy`)

`3/syntax-error-easy/fb.txt`	The feedback file for this exercise.
`3/syntax-error-easy/error1.c`	Working code for missing semicolon example.
`3/syntax-error-easy/error2.c`	Working code for missing quotation mark example.
`3/syntax-error-easy/error3.c`	Working code for invalid variable name example.
`3/syntax-error-easy/error4.c`	Working code for missing newline character example.
`3/syntax-error-easy/error5.c`	Working code for broken string literal example.

Task 2: Compiler/Syntax Errors (6 Marks)

This task addresses more complex programs that contain multiple errors and doesn't provide a description of the encountered errors.

It is again your task to repair the programs such that they compile and produce the intended output. Make sure that the compilation does not return any warning!

Note that there might be some syntactical constructs that you are not yet familiar. However, to fix them, you should not need additional knowledge.

The three files are as follows:

You can also find them in CSGitLab: `https://csgitlab.reading.ac.uk/di918039/cs1pr-portfolio/tree/master/3/syntax-error`.

Submission (directory: `3/syntax-error`)

`3/syntax-error/fb.txt`	The feedback file for this exercise.
`3/syntax-error/fixed1.c`	Compilable program for fixed1.c
`3/syntax-error/fixed2.c`	Compilable program for fixed2.c
`3/syntax-error/fixed3.c`	Compilable program for fixed3.c

Hints

- An error may generate subsequent (parsing) errors that are actually caused by the initial error. Always attempt to fix the first error, then move on.

Task 3: Counting the line-number of the input (5 Marks)

This tutorial aims to apply your C programming skills to the task of creating a practical tool. We want to replicate the behaviour of the `wc -l` command, which will return the line count of a file passed to its standard input.

Example output after running the command on a file with 13 lines:

```
 $ cat file | wc -l
 13, file
```

The C standard library contains the function `getchar()`, which is very useful for a program such as this, as it returns one character at a time from the standard input (stdin) until no characters are left. That means that the program can simply use a while loop to retrieve characters until the standard input stream is empty. You do not have to understand how functions work, just assume here this statement returns the character needed.

The pseudocode for this program is included below:

```
lineCount = 0
currentCharacter = getchar()
while currentCharacter != EOF { // As long as the current character isn't EOF, iterate
  if currentCharacter == '\n' {
    lineCount = lineCount + 1
  }
  currentCharacter = getchar()
}
print(lineCount)
```

Next, you can find a C template file for the line-count program which you can use to get started:

```c
#include <stdio.h>

void main() {
  // Declare two variables to store the line counter and the current character (this works)
  int lineCount = 0;
  char c;
  // Now create a while loop to retrieve characters using getchar() and count the lines

  // If statement to check if the current character is a new line

  // Print the line counter to stdout:
  printf("Line Counter: %d\n", lineCount);
}
```

Given a file with 13 lines, we expect the following output:

```
$ cat file | ./linecount
Line Counter: 13
```

Given the previous instructions, have a go at the following steps:

Steps

1. Open the Atom text editor
2. Create a new file called `linecount.c`
3. Write the line-count program in the source code file by translating the remaining pieces of the pseudocode into C.
4. Write a shell script which compiles and runs the source code using the source file as the input
5. Check that the program outputs the correct line count

Submission (directory: `3/linecount`)

`3/linecount/fb.txt`	The feedback file for this exercise.
`3/linecount/linecount.c`	Source file containing your line counting program. It shall not output anything else than what is already written by the statement in Line 12.
`3/linecount/compile.sh`	Bash script that compiles and runs your program.

Hints

- The man pages provide also help for the C programming interface. Just type `$ man 3 <FUNCTION>`, for example `$ man 3 getchar` or `$ man 3 atoi`. The manpages contain a lot of information that can be overwhelming, focus on the description section and return value if you want to read it. The man pages are organised into chapters, and chapter 3 covers the programming.
- EOF stands for End Of File. It represents when the program can stop reading from standard input and finish the execution of the while loop. If the program never encounters an EOF character, it will loop infinitely. Perhaps an improvement to be considered might be adding a case where the program stops if the file is too large?

Task 4: Implementing Algorithms for Binary Arithmetic (15 Marks)

In this task, you have to develop algorithms for the operations of a primitive calculator for binary numbers.

The calculator should manipulate 8-bit binary numbers $[b_7, ..., b_0]_2$ providing the following operations:

- Addition
- Subtraction
- Multiplication (OPTIONAL! If you don't feel challenged with the previous operations, do it!)

Steps

1. Create a single program that is able to perform the user selected operation on two decimal numbers provided as two input arguments. The internal calculations and the output of the program must be in

binary.

2. Sketch the flow graph for the addition operation.
3. Provide a shell script to compile the program.

Synopsis

```
$ ./calc <Number1> <op> <Number2>
```

Example execution

```
$ ./calc 3 "+" 7
```

Expected Output:

```
00001010
```

We will use a `for` loop to iterate a given number of times (details will be taught on Thursday). In this example, we must iterate over all bits. A skeleton is provided for your convenience that performs a trivial check and uses the `for` loop to print four values.

Skeleton code:

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char ** argv){
  // read the arguments provided by the user, you do not have to understand how the next three lines work.
  int dec1 = atoi(argv[1]);
  char op = argv[2][0]; // +, *, -, ...
  int dec2 = atoi(argv[3]);

  // print the arguments, for you to test, remove it before you submit!
  printf("Called with dec1: %d op: %c dec2: %d\n", dec1, op, dec2);

  // check if a bit is set, you need this when you perform the addition in binary!
  if(dec1 & 1){
    // bit with value 1 is set in dec1
    printf("bit is 1\n");
  }

  // make a decision based on the type of the operation
  if (op == '+'){
    // our iteration construct: think about how many times you must repeat something for the addition
    for (int i=0; i < 4; i++){
      // do something, e.g., add the different bits
      printf("%d\n", i); // here we print the value of i
    }
  }
  printf("\n");
  return 0;
}
```

Submission (directory: `3/binary-calculator`)

`3/binary-calculator/fb.txt`	The feedback file for this exercise.
`3/binary-calculator/compile.sh`	Bash script that compiles the program.
`3/binary-calculator/calc.c`	The C-program for your calculator.
`3/binary-calculator/flow-graph-addition.pdf`	The flow graph for your add program.

Hints

- Compile the program using `$ gcc -Wall calc.c -o calc` and run it with `$ ./calc 3 + 5`. It should output a string (Line 11 and the other messages)...
- Apply 2's complement to represent negative numbers.
- Use bitwise manipulation and bit shifting to extract individual bits and add them together.
- It might be useful to provide some tests (using the test harness) to check that your program works correctly.

Further Reading

- `https://www.tutorialspoint.com/computer_logical_organization/binary_arithmetic.htm`