

COSC2430 — Spring 2021

Terminal control, enumerations, headers, separate compilation, Makefile

1. Terminal control escape sequences

```
echo -e '\a'
```

The echo command just prints a sequence of characters to the screen. The -e option enables the interpretation of some backslash-escaped characters. For example, if you type

```
echo "\0110\0165\0156\0147"
```

then the same string is printed. On the other hand, with the -e option you get

```
echo -e "\0110\0165\0156\0147"
```

Here, 110 is the ASCII code of letter H (in decimal). The escape sequence tells echo to interpret the number that follows in decimal format. You could have achieved the same by using the hexadecimal format: \x48 is the hex format of letter H. Try some of the following commands to see the effect of terminal control sequences:

```
echo -e 'Make it \033[31mred\033[39m, \033[33myellow\033[39m, or \033[05mblinking\033[05m'
```

```
echo -e 'Change \033[41mbackground and \033[49mreset'
```


```
echo -e '\033[2J'
```

The rule is pretty simple. Say we want to change the foreground color to yellow, we output the following sequence of characters consecutively: the ESC character (33 in oct, `0x1b` in hex, 27 in dec), followed by `[33m`. If we want to reset the foreground color to its original setting, output ESC followed by `[39m`. The ESC character tells the terminal to interpret the following characters as (color) control codes. Some of the more commonly used (<http://wiki.bash-hackers.org/scripting/terminalcodes>) codes are:


The text attributes

ANSI	Description
<code>[0 m</code>	Reset all attributes
<code>[1 m</code>	Set "bright" attribute
<code>[2 m</code>	Set "dim" attribute
<code>[4 m</code>	Set "underscore" (underlined text) attribute
<code>[5 m</code>	Set "blink" attribute
<code>[7 m</code>	Set "reverse" attribute
<code>[8 m</code>	Set "hidden" attribute

Foreground coloring

ANSI	terminfo equivalent	Description
<code>[0 m</code>	Set foreground to color #0 – black	
<code>[1 m</code>	Set foreground to color #1 – red	
<code>[3 2 m</code>	Set foreground to color #2 – green	
<code>[3 3 m</code>	Set foreground to color #3 – yellow	
<code>[3 4 m</code>	Set foreground to color #4 – blue	
<code>[3 5 m</code>	Set foreground to color #5 – magenta	
<code>[3 6 m</code>	Set foreground to color #6 – cyan	
<code>[3 7 m</code>	Set foreground to color #7 – white	
<code>[3 9 m</code>	Set default color as foreground color	

Background coloring

ANSI	terminfo equivalent	Description
<code>[4 0 m</code>	Set background to color #0 – black	
<code>[4 1 m</code>	Set background to color #1 – red	
<code>[4 2 m</code>	Set background to color #2 – green	
<code>[4 3 m</code>	Set background to color #3 – yellow	
<code>[4 4 m</code>	Set background to color #4 – blue	

[4 5 m	Set background to color #5 – magenta
[4 6 m	Set background to color #6 – cyan
[4 7 m	Set background to color #7 – white
[4 9 m	Set default color as background color

2. Enumerations and function default arguments

We want to write a function named `term_cc()` that takes three arguments: a foreground color, a background color, and an attribute parameter following the codes in the tables above. Each of these arguments can be thought of as a member of a small finite set. So, we ideally would like to define a couple of new types: color type and attribute type, and use them as parameters of the function. Enumerations (http://www.cplusplus.com/doc/tutorial/other_data_types/) can be used to create new data types which are typically used to represent variables that can only take values from a fixed finite set. The function `term_cc()` can be written as follows.

```

1  // =====
2  // colors.cpp
3  // ~~~~~
4  // author: hqn
5  // + illustrate terminal control codes
6  // + illustrate a user defined type using the 'enum' construct
7  // + illustrate typical "sections" of a source cpp file that contains main()
8  // + note that what I did in this file is overkill for a source file this size
9  //   I just wanted to illustrate typical programming style, which is always a
10 //   guideline, not an absolute standard to follow all the time. However, IMHO
11 //   it is important to develop good programming habit even for small programs
12 // =====
13
14 #include <iostream>
15 #include <sstream>
16 #include <string>
17 using namespace std; // BAD PRACTICE
18
19 // ~~~~~
20 // Typically Part 1 of a big cpp file contains combinations of
21 // + #include statements,
22 // + typedefs,
23 // + global variables,
24 // + function prototypes
25 // ~~~~~
26
27 // the terminal's attribute type
28 enum term_attr_t {
29     DEFAULT_ATTRIB = '0'
30     BRIGHT        = '1',
31     DIM           = '2',
32     UNDERLINE    = '4',
33     BLINK        = '5',
34     REVERSE       = '7',
35     HIDDEN        = '8',
36 };

```

```

37
38 // the colors, background or foreground
39 enum term_colors_t {
40     BLACK    = '0',
41     RED      = '1',
42     GREEN    = '2',
43     YELLOW   = '3',
44     BLUE     = '4',
45     MAGENTA  = '5',
46     CYAN     = '6',
47     WHITE    = '7',
48     DEFAULT_COLOR = '9'
49 };
50
51 // term_cc() outputs the corresponding terminal control escape sequence (TCES)
52 string term_cc(term_colors_t fg=DEFAULT_COLOR,
53               term_colors_t bg=DEFAULT_COLOR,
54               term_attr_t attr=DEFAULT_ATTRIB);
55
56 // ~~~~~
57 // Typical Part 2:
58 // + the main function itself
59 // this should be short and sweet, illustrate the key skeleton of the program
60 // if you write a very long main(), break up functionalities into smaller
61 // functions for readability
62 // ~~~~~
63
64 /**
65  * -----
66  * this program tests our terminal control routine term_cc
67  * -----
68  */
69 int main()
70 {
71     cout << term_cc(YELLOW) << "I'm yellow!\n"
72          << term_cc() << "I'm normal!" << endl
73          << term_cc(BLUE, YELLOW, BLINK) << "I'm blinking blue on yellow!\n"
74          << term_cc() << "I'm back to normal!\n";
75     return 0;
76 }
77
78 // ~~~~~
79 // Typical Section 3: function definitions
80 // ~~~~~
81
82 /**
83  * -----
84  * output a terminal control code string which formats the text background
85  * & foreground colors & text attributes
86  * usage:
87  *   cout << term_cc(YELLOW) << "This text is yellow\n" << term_cc();
88  *   cout << term_cc(BLUE, YELLOW, BLINK) << "I'm blinking blue on yellow!\n"
89  *
90  * notice the type casts: char(fg), char(bg)
91  * -----
92  */
93 string term_cc(term_colors_t fg, term_colors_t bg, term_attr_t attr)

```

```

94 {
95     ostream oss;
96     oss << "\033[" << char(attr) << ";3" << char(fg) << ";4" << char(bg) << 'm
97     return oss.str();
98 }

```

There are several new concepts here.

- A variable such as `fg` which is of an `enum` type can only take one of the enumerated values represented by the names `BLACK`, `WHITE` etc.
- The names inside an `enum` are actually *integer constants*. Remember that character `'0'` has integral value 48 (its ASCII code (<http://www.ascii-code.com/>)), character `'1'` has integral value 49 and so on.
- If we don't care about the actual values of these constants, we don't have to specify them. For example, it is perfectly fine to do this:

```
enum months_t { JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };
```

And, we certainly don't have to capitalize the names inside. This is just my (bad) habit from years of programming in C. If we don't assign values to the names, the compiler will do it for us; typically they are assigned with non-negative integers 0, 1, ... sequentially. Although the values are integers, we can't assign integers directly to `enum` variables, we have to use the specified names. For example,

```
enum my_type_t { A, B, C };
```

```

int main() {
    my_type_t x = A;
    int y = x;           // OK because enum is integer-like
    cout << "x = " << x << endl; // x = 0
    cout << "y = " << y << endl; // y = 0
    x = 0;                // this is a compilation error
    return 0;
}

```

- In the body of function `term_cc` there are several type casts (<http://www.cplusplus.com/doc/tutorial/typecasting/>), which tell the compiler to specifically interpret the (integer) values `attr`, `fg`, and `bf` as of type `char`.
- The prototype

```
string term_cc(term_colors_t fg=DEFAULT_COLOR,
               term_colors_t bg=DEFAULT_COLOR,
               term_attr_t attr=DEFAULT_ATTRIB);
```

makes use of default arguments

([http://publib.boulder.ibm.com/infocenter/comphelp/v7v91/index.jsp?](http://publib.boulder.ibm.com/infocenter/comphelp/v7v91/index.jsp?topic=%2Fcom.ibm.vacpp7a.doc%2Flanguage%2Fref%2Fclrc07cplr237.htm)

[topic=%2Fcom.ibm.vacpp7a.doc%2Flanguage%2Fref%2Fclrc07cplr237.htm](http://publib.boulder.ibm.com/infocenter/comphelp/v7v91/index.jsp?topic=%2Fcom.ibm.vacpp7a.doc%2Flanguage%2Fref%2Fclrc07cplr237.htm)). Hence, if we call the function with no parameter the default values are used. If we call it with one argument, it will be interpreted as the `fg` value; if we call the function with two arguments, they are understood as `fg`, `bg`.

- In the definition of `term_cc`, we made use of an ostream object (http://en.cppreference.com/w/cpp/io/basic_ostream), to which we can do `cout`-like operations

on. The difference is that inserting a character stream into `cout` (typically) prints the stream to the screen, while inserting a character stream into an `ostream` object will store the stream in the object as a character sequence which we can then get using the `str()` member function.

3. Separate compilations and headers

3.1. Physical separation of source codes

Beside changing colors, we can also write other terminal control routines such as “ring a bell,” “clear the screen,” or a hundred other commands. These terminal control routines should be useful in many different terminal-based programs we write. Hence, it is desirable to write a little “terminal control library”, test it carefully, and reuse the library without touching the library’s source codes or recompiling the library. In C++, we can accomplish this task by creating two new files: `term_control.cpp` and `term_control.h` (the header file).

The header file contains the “interface” to the library. The interface typically consists of type definitions, function prototypes, class and struct definitions, `#define` macros, `#pragma` directives, template and inline functions, and constants (though we should try to avoid global constants as much as possible).

```

1 // =====
2 // term_control.h
3 // ~~~~~
4 // author : hqn
5 // - a few functions which output terminal control escape sequences
6 // =====
7
8 #ifndef TERM_CONTROL_H_
9 #define TERM_CONTROL_H_
10
11 #include <string>
12
13 // the attributes
14 enum term_attrib_t {
15     DEFAULT_ATTRIB = '0',
16     BRIGHT        = '1',
17     DIM           = '2',
18     UNDERLINE    = '4',
19     BLINK         = '5',
20     REVERSE       = '7',
21     HIDDEN        = '8'
22 };
23
24 // the colors, background or foreground
25 enum term_colors_t {
26     BLACK = '0',
27     RED   = '1',
28     GREEN = '2',
29     YELLOW = '3',
30     BLUE  = '4',
31     MAGENTA = '5',

```

```

32     CYAN      = '6',
33     WHITE    = '7',
34     DEFAULT_COLOR = '9'
35 };
36
37
38 /**
39  * -----
40  * term_cc() outputs the corresponding terminal control escape sequence (TCES)
41  *
42  * usage:
43  * cout << term_cc(YELLOW) << "This text is yellow\n" << term_cc();
44  * cout << term_cc(BLUE, YELLOW, BLINK) << "I'm blinking blue on yellow!\n"
45  * cout << term_fg(YELLOW) << "This text is yellow\n" << term_cc();
46  *
47  * term_clear() outputs the TCES which clears the screen
48  * -----
49  */
50 std::string term_cc(term_colors_t fg=DEFAULT_COLOR,
51                    term_colors_t bg=DEFAULT_COLOR,
52                    term_attr_t attr=DEFAULT_ATTRIB);
53 std::string term_bg(term_colors_t bg=DEFAULT_COLOR);
54 std::string term_fg(term_colors_t fg=DEFAULT_COLOR);
55 std::string term_attr(term_attr_t attrib=DEFAULT_ATTRIB);
56 std::string term_clear();
57
58 #endif // end of #ifndef TERM_CONTROL_H_

```

☐ **body” file** typically contains the definitions of the functions that the library provides.

```

1  // =====
2  // term_control.cpp
3  // ~~~~~
4  // author : hqn
5  // - implementations of a few functions which return a string of
6  //   terminal control escape sequences
7  // =====
8  #include <sstream>
9  #include <iostream>
10 #include "term_control.h"
11
12 /**
13  * -----
14  * output a terminal control code string that formats the text background
15  * -----
16  */
17 std::string term_bg(term_colors_t bg)
18 {
19     std::ostringstream oss;
20     oss << "\033[4" << char(bg) << 'm';
21     return oss.str();
22 }
23
24 /**
25  * -----
26  * output a terminal control code string that formats the text foreground

```

```

27  * -----
28  */
29  std::string term_fg(term_colors_t fg)
30  {
31      std::ostringstream oss;
32      oss << "\033[3" << char(fg) << 'm';
33      return oss.str();
34  }
35
36  /**
37  * -----
38  * output a terminal control code string that formats the text attribute
39  * -----
40  */
41  std::string term_attr(term_attr_t attr)
42  {
43      std::ostringstream oss;
44      oss << "\033[" << char(attr) << 'm';
45      return oss.str();
46  }
47
48  /**
49  * -----
50  * output a terminal control code string which formats the text background
51  * & foreground colors & text attributes
52  * -----
53  */
54  std::string term_cc(term_colors_t fg, term_colors_t bg, term_attr_t attr)
55  {
56      std::ostringstream oss;
57      oss << "\033[" << char(attr) << ";3" << char(fg) << ";4" << char(bg) << 'r';
58      return oss.str();
59  }
60
61  /**
62  * -----
63  * output a TCES which clears the screen
64  * -----
65  */
66  std::string term_clear()
67  {
68      return "\033[2J";
69  }

```

In order to use the library (or module), we include the header file which contains the declarations we needed.


```

1  // tcdriver.cpp
2  #include <iostream>
3  #include <string>
4
5  #include "term_control.h"
6
7  int main() {
8      std::cout << term_cc(YELLOW) << "I'm yellow\n"
9          << term_cc() << "I'm back" << std::endl
10         << term_cc(BLUE, YELLOW, BLINK) << "I'm blinking blue on yellow!\n"
11         << term_cc() << "I'm back!\n";
12      std::cout << "Enter anything: ";
13      std::string line;
14      getline(std::cin, line);
15      std::cout << term_clear() << "Clear screen, bye\n";
16      return 0;
17  }

```

To compile `tcdriver.cpp` into an executable, one option would be to do

```
g++ tcdriver.cpp term_control.cpp
```

The compiler is smart enough to look for the only `main()` function from the list of source files. The effect is pretty much the same as if we were dumping all `cpp` files into `tcdriver.cpp`. This option is not ideal. If we used the terminal control routines in another program then we have to recompile the `term_control.cpp` again. Imagine we are writing a lot of programs using not only the terminal control “library” but also other libraries which we have developed over the years. It would be nice to be able to compile `term_control.cpp` once and then reuse this compilation later. This is exactly the situation with C++’s standard library. We include `iostream` which is a header file, whose body is already compiled and can be used by each one of the students in this course.

3.2. Separate compilation and the g++ compilation process

Very roughly, `gcc/g++` goes through four stages (http://qcd.phys.cmu.edu/QCDcluster/gnu/g++_man.html) of “compilation”: preprocessing, compiling, assembling, and linking.

1. It calls the preprocessor (`cpp`) to preprocess the source files: dumping header files into the body, expand macros, remove comments, etc. The output of this stage is a `.i`, `.ii` file. Try `g++ -E hello_world.cpp`.
2. Then the actual compilation takes place, where the preprocessed source file(s) is compiled into the target architecture’s assembler code file(s). Try `g++ -S hellow_world.cpp`, which will produce a `.s` file.
3. Then it calls `as` to assemble the assembly code into object code. Try `g++ -c hello_world.cpp`. Object codes are stored in `.o/.obj` files, which are machine codes but not executable yet.

4. Finally, libraries and object codes are linked together to obtain the executable. Try `g++ hello_world.o`.

Back to the terminal control example, it is easy to compile a separate source file:

```
g++ -c term_control.cpp
```

The `-c` option (<http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Overall-Options.html#Overall%20Options>) tells the compiler to compile the functions in `term_control.cpp` and store the (almost) machine codes in an output file `term_control.o`. (The option also tells the compiler to not link just yet.) Then, to compile `tcdriver.cpp` we can type

```
g++ tcdriver.cpp term_control.o
```

So, if we have a second “client” `test2.cpp` of the terminal control library we can do

```
g++ test2.cpp term_control.o
```



and so on. The terminal control library does not need to be recompiled. A client of the library can simply include the header file that contains all the necessary declarations and definitions. By not giving out the source code in `term_control.cpp`, we achieve a crude form of information hiding (<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/1996/N0885.pdf>). Even if terminal control is only used in the current program we are writing, it is still a good idea to physically partition the source codes into different modules, each of which represents some logical unit of functionality: (1) we will save compilation time by not recompiling, and (2) we can do unit-testing with parts of the program we are developing.

3.3. The header file and the `#include` guard

In `term_control.h` there are three curious lines:

```
1  #ifndefn TERM_CONTROL_H_
2  #define TERM_CONTROL_H_
3
4  // typedefs and prototypes go here
5
6  #endif
```

The words `#ifndef`, `#define`, `#endif` are called preprocessor directives (<http://publib.boulder.ibm.com/infocenter/comphelp/v8v101/index.jsp?topic=%2Fcom.ibm.xlcpp8a.doc%2Flanguage%2Fref%2Fpcl.htm>). In fact, we have been using `#include` which is also a preprocessor directive. What happens is that before actually compiling a source program, the compiler calls a special program/routine called the preprocessor (<http://en.cppreference.com/w/cpp/preprocessor>) to process the input files before compiling. The preprocessor directives are essentially commands for the preprocessor. The `#include` commands tells the preprocessor to dump the entire included file (the header file) into the source file (the cpp file), for example.

The `#ifndef`, `#define`, `#endif` directives say that if the identifier `TERM_CONTROL_H_` is not defined then we define it and also does the rest of the task in the body of the header file. On the other hand, if the identifier `TERM_CONTROL_H_` is defined then effectively the header is empty. This construct is called an include guard (http://en.wikipedia.org/wiki/Include_guard) which prevents multiple inclusions of the same header. In a big program with many source files, a source file may include several header files, some of which might contain circular or duplicate inclusions out of necessity. For example, we might include in `main.cpp` two header files `module1.h` and `module2.h`, and each of those header files includes `module3.h`. Then, effectively `main.cpp` has 2 inclusions of `module3.h`. If there is no include guard in `module3.h`, we might encounter a compilation error due to the duplications of definitions in `module3.h`. The include guard construct helps prevent compilation errors. We are using the convention (http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#The_define_Guard) in Google's style guide for naming the guard.

Another method for preventing multiple inclusion is to use something called #pragma once method (http://en.wikipedia.org/wiki/Pragma_once). This method is simpler, but it is not standard-conforming ☐ thus not guaranteed to be supported by compilers.

System header files such as `iostream.h` or `string.h` are included using the syntax `#include <iostream>`, `#include <string>`; while user-defined header files are included using the syntax `#include "term_control.h"`.

Last but not least, we explicitly mentioned the returned type of the functions in `term_control.h` as `std::string`. We could have inserted a `using namespace std;` line on top and refer to `string` directly without the prefix `std::`; however, that means any file which includes `term_control.h` will implicitly have the `using namespace std;` line and that might create unnecessary name conflicts. It is a good programming practice to not use an entire namespace in the headers.

For more information on splitting the source into multiple files, and separate compilation using Makefile, see

- Organizing code files in C/C++ (http://www.gamedev.net/page/resources/_/technical/general-programming/organizing-code-files-in-c-and-c-r1798).
- A short Makefile tutorial (<http://www.cs.umd.edu/class/spring2002/cmsc214/Tutorial/makefile.html>).
- GCC and Make, Compiling, Linking, and Building C/C++ applications (http://www.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html).

Please read the Makefile tutorial. It will save you a lot of time later in the course. Following the tutorial, we have the following Makefile.

```
# Makefile
OBS = term_control.o tcdriver.o
CC = g++
DEBUG = -g
CFLAGS = -Wall -c $(DEBUG)
LFLAGS = -Wall $(DEBUG)

ttc : $(OBS)
    $(CC) $(LFLAGS) $(OBS) -o ttc

term_control.o : term_control.h term_control.cpp
    $(CC) $(CFLAGS) term_control.cpp

tcdriver.o : term_control.h tcdriver.cpp
    $(CC) $(CFLAGS) tcdriver.cpp

clean:
    rm -f *.o ttc
```

