Constants and State Machines in C

This lecture covers the use of symbolic constants in C programs and the structure of C programs that implement state machines.
After this lecture you should be able to use symbolic constants in C programs and write a C program to implement a state machine.

Symbolic Constants

Most programs use constants. Examples of constants include I/O port addresses, bit patterns ("masks"), array lengths, string values, etc.

Putting constants directly into the statements in a program makes it difficult to find and change the values of these constants if the program needs to be modified. Embedding constants directly into the code can also be confusing because, unlike variables, constants don't have names that can help explain their meaning.

The solution to these problems is to use symbols similar to variable names to represent constants. In C this is done by using `#define` lines. Each such line defines the value of a symbol. Wherever the defined symbol appears in the rest of the program the compiler will replace the symbol with the previously defined value.

For example, if we had the following lines in a program:

```
#define IOPORT  32
#define MSG     "Hello, world!\n"
#define u_char  unsigned char
#define NCHR    8
```

then whenever the symbols `IOPORT`, `MSG`, `u_char` or `NCHR` appeared anywhere in a subsequent line in the program they would be replaced by corresponding text. The substitution takes place before the line is seen by the rest of the compiler. This allows the constants to be of any type (numbers, strings or even keywords). Thus we could have lines in the program such as:

```
int x[NCHR] ;
u_char c2 ;
spoke ( IOPORT, 1 ) ;
printf ( "%s", MSG ) ;
```

Typically the `#defines` are placed at the start of a source file, immediately after the `#include` lines.

It is good practice[1] to use symbolic constants for all constants in a program. The only exceptions should be cases where the purpose of the constant is clear from the context. The only common exceptions are the values 0 (e.g. `i=0`) and 1 (e.g. `i=i+1`). Constants with other values should almost always be replaced by symbolic constants.

State Machines in C

We have seen that state machines allow us to unambiguously describe the behaviour of a controller. One way to implement a state machine is by writing a program that behaves the same way as a state machine.

Such a program is written as a single continuous (infinite) loop. In each iteration through the loop the program does two things: (1) selects the next state according to the current state and the values of the inputs, and (2) sets the outputs according to the current state.

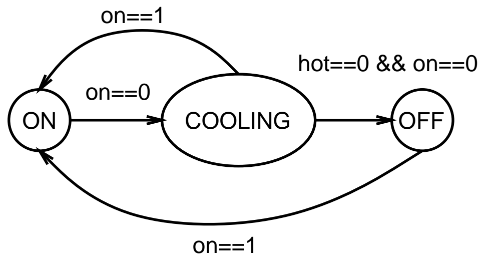
A a program that implements a state machine can be written as follows:

- assign a unique number to each state
- define an integer variable whose value will be the current state
- use an infinite loop that encloses the remaining statements:
- use `if/else` statements to select a different set of statements to be executed depending on the current state
- write statements for each state that set the outputs

[1]You will be expected to use symbolic constants in future assignments and labs whenever it is appropriate.

- write statements for each state that set the next state based on the current input

For example, the state transition diagram for the fan motor controller state machine given in the lecture on state machines was:



Here is an example of how we would write the code for this state machine:

```
#define ON 1
#define COOLING 2
#define OFF 3

main()
{
  int on, hot, run, state ;

  while ( 1 ) {

    if ( state == ON ) {

      run = 1 ;

      if ( on == 0 ) {
        state == COOLING ;
      }

    } else if ( state == COOLING ) {

      run = 1 ;

      if ( on == 0 && hot == 0 ) {
        state == OFF ;
      } else if ( on == 1 ) {
        state = ON ;
      }

    } else if ( state == OFF ) {

      run = 0 ;

      if ( on == 1 ) {
        state = ON ;
      }

    }

  } /* end of while() loop */

}
```

Note that the structure of the program will be the same for every state machine. The only changes will be in the number of states and in the statements that set the next state and the output values.

Exercise: Consider a state machine that implements a three-digit digital combination lock. The lock should open when the digits 3, 7, and 4 are entered sequentially and close when a wrong digit is entered. Draw the state transition diagram. Write the C code that implements this state machine assuming you have functions `getch()` and `setlock(int locked)` available.