This article demonstrates the state machine's implementation in the C programming language.

An Overview of State Machine

Using a state machine to implement code is a valuable design strategy to resolve complex engineering issues. State machines take the overall design and split it into stages, referred to as states inside the state-machine jargon.

Every state is responsible for carrying out a particular function. On the other side, events are the stimuli that cause the state machine to change between states. It is also referred to as a `transition`.

When first written, most systems are straightforward and well-organized, but when new features are added, additional flags and variables are created to keep track of events' history.

Then, `if` and `else` statements are added to test the ever-more-complex logical expressions created out of the many variables and flags.

In this regard, the state machines are of assistance. When utilized appropriately, state machines simplify the conditions tested at each branching point and make it easier to `switch between various modes of program execution`.

It turns out that the behaviour of most real-time systems can be divided into a relatively small number of non-overlapping chunks (states).

The event responses within each chunk depend only on the current event, and they no longer rely on the sequence of events that have occurred in the past.

Use `Function Pointers` to Create a State Machine

The following is an example of code to build a state machine in C. This specific example generates two states switches back and forth between them ten times.

Then, it displays the index number and the state corresponding to that index.

```c
#include <stdio.h>


struct state;
typedef void state_fn(struct state *);


struct state {
  state_fn *next;
  int i;
};


state_fn off_state, on_state;


void off_state(struct state *state) {
  printf("%s %i\n", __func__, ++state->i);
  state->next = on_state;
}


void on_state(struct state *state) {
  printf("%s %i\n", __func__, ++state->i);
  state->next = state->i < 10 ? off_state : 0;
}


int main(void) {
  struct state state = {off_state, 0};
  while (state.next) state.next(&state);
}
```

Output:

```
off_state 1

on_state 2

off_state 3

on_state 4

off_state 5

on_state 6

off_state 7

on_state 8

off_state 9

on_state 10
```

Use `Switch` Statement to Create a State Machine

To get things off to a solid start, you should list the set of `states` that our program may be in and the set of `events` it can handle.

This program may be in one of these three states: `START`, `ITERATE` or `END`.

```
enum states {

  START,

  ITERATE,

  END,

} state;
```

It can handle the events `START LOOPING`, `SHOW_MESSAGE`, and `STOP LOOPING`. After receiving the `START LOOPING` event while in the `START` state, our program will transit into the `ITERATE` state.

While in the `ITERATE` state, it will show the message when it receives the `SHOW_MESSAGE` event. Finally, it will transition into the `END` state after receiving the `STOP LOOPING` event.

```
enum events {

  START_LOOPING,

  SHOW_MESSAGE,

  STOP_LOOPING,

};
```

Let's use a `switch` statement that looks at the current state and executes the appropriate case. We have three primary situations, which are referred to as `START`, `ITERATE` and `END`.

Inside the `START` case, if the event received is `START LOOPING`, it will change the state to `ITERATE`. The case will be broken if the event is not `START LOOPING`.

Similarly, within the `ITERATE` case, if the event received is `SHOW MESSAGE`, it will display the message; otherwise, it will check for the `STOP LOOPING` case and change the state to `END`.

If none of these events is received, it will break and go on to the next case. Finally, the `END` case causes the `switch` statement to terminate, and the program exits.

```
void switchState(enum events event) {
  switch (state) {
    case START:
      switch (event) {
        case START_LOOPING:
          state = ITERATE;
          break;
        default:
          exit(1);
          break;
      }
      break;
    case ITERATE:
```

```
      switch (event) {
        case SHOW_MESSAGE:
          printf("State Machine Ready!\n");
          break;
        case STOP_LOOPING:
          state = END;
          break;
        default:
          exit(1);
          break;
      }
      break;
    case END:
      exit(1);
      break;
  }
}
```

At last, let us invoke the method known as `switchState()` and provide it with the events one at a time in the form of arguments.

```
int main(void) {

  switchState(START_LOOPING);

  switchState(SHOW_MESSAGE);

  switchState(STOP_LOOPING);

  return 0;

}
```

Following is the complete source code using the `switch` statement.

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

enum states {
  START,
  ITERATE,
  END,
} state;

enum events {
  START_LOOPING,
  SHOW_MESSAGE,
  STOP_LOOPING,
};
```

```c
void switchState(enum events event) {
  switch (state) {
    case START:
      switch (event) {
        case START_LOOPING:
          state = ITERATE;
          break;
        default:
          exit(1);
          break;
      }
      break;
    case ITERATE:
      switch (event) {
        case SHOW_MESSAGE:
          printf("State Machine Ready!\n");
          break;
        case STOP_LOOPING:
          state = END;
          break;
        default:
          exit(1);
          break;
      }
      break;
    case END:
      exit(1);
      break;
  }
}

int main(void) {
  switchState(START_LOOPING);
  switchState(SHOW_MESSAGE);
  switchState(STOP_LOOPING);
  return 0;
}
```

Output:

```
State Machine Ready!
```