Embedded Systems Fundamentals

ENGD2103

Dr M A Oliver

michael.oliver@dmu.ac.uk

Lecture 5: Classes

Contents

This lecture will include:-

- Setting the scene
  - where we left off
- Classes:
  - Recap on C data structure
  - Use of C++ to introduce classes
- The use of classes for concurrent operations
  - First improvements for improving structure.

Concurrency: Where we currently stand

```
#include "hal.h"

bool init_module0_clock;
bool init_module1_clock;
bool init_module2_clock;


void setup() {
  HAL_gpioInit();
  init_module0_clock = true;
  init_module1_clock = true;
  init_module2_clock = true;
}

void loop() {
  // put your main code here, to run repeatedly:

  { // module 0
    static unsigned long module_time, module_delay;
    static bool module_doStep;
    static unsigned char state; // state variable module 0

    if (init_module0_clock) {
      module_delay = 500;
      module_time = millis();
      module_doStep = false;
      init_module0_clock = false;
      state=0;
    }
    else {
      unsigned long m = millis();
      if ( (m - module_time) > module_delay ) {
        module_time = m;
        module_doStep = true;
      }
      else module_doStep = false;
    }

    if (module_doStep)
    {
      switch(state)
      {
      case 0:
        HAL_ledRed1On;
        state = 1;
        break;
      case 1:
        HAL_ledRed1Off;
        state = 0;
        break;
      }
    }
  }

  { // module 1
    static unsigned long module_time, module_delay;
    static bool module_doStep;
    static unsigned char state; // state variable module 1

    if (init_module1_clock) {
      module_delay = 300;
      module_time = millis();
      module_doStep = false;
      init_module1_clock = false;
      state=0;
    }
    else {
      unsigned long m = millis();
      if ( (m - module_time) > module_delay ) {
        module_time = m;
        module_doStep = true;
      }
      else module_doStep = false;
    }

    if (module_doStep)
    {
      switch(state)
      {
      case 0:
        HAL_ledYellow1On;
        state = 1;
        break;
      case 1:
        HAL_ledYellow1Off;
        state = 0;
        break;
      }
    }
  }

  { // module 2
    static unsigned long module_time, module_delay;
    static bool module_doStep;
    static unsigned char state; // state variable module 2

    if (init_module2_clock) {
      module_delay = 500;
      module_time = millis();
      module_doStep = false;
      init_module2_clock = false;
      state=0;
    }
    else {
      unsigned long m = millis();
      if ( (m - module_time) > module_delay ) {
        module_time = m;
        module_doStep = true;
      }
      else module_doStep = false;
    }

    if (module_doStep)
    {
      switch(state)
      {
      case 0:
        HAL_ledGreen1On;
        module_delay = 600;
        state = 1;
        break;
      case 1:
        HAL_ledGreen1Off;
        module_delay = 450;
        state = 2;
        break;
      case 2:
        HAL_ledGreen1On;
        module_delay = 1800;
        state = 3;
        break;
      case 3:
        HAL_ledGreen1Off;
        module_delay = 1200;
        state = 0;
        break;
      }
    }
  }
```

Code for blinking 3 LEDs independently and concurrently….

Concurrency: Where we currently stand

PROS

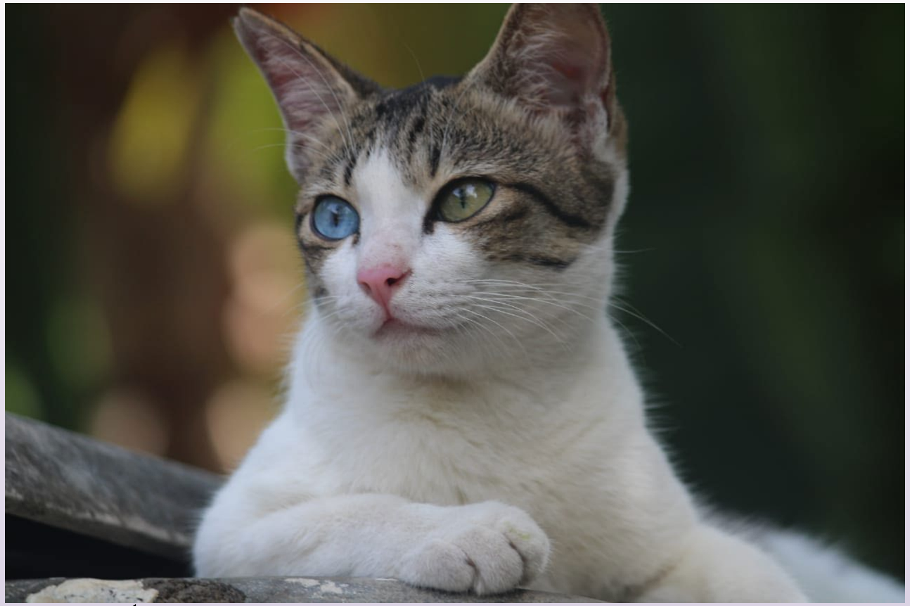
- We have an approach that works well.
- We have an approach that is easy to implement.

CONS

- The resultant code is unstructured
- It is very difficult to follow and maintain.

We need a cleaner, more structured approach.

Introduction to Classes



Consider a cat!

- It can be characterized by many parameters
- Consider just two numeric parameters
  - One could be age (in years) – represented by an integer
  - One could be weight (in kg) – represented by a floating point value
- Having all these properties as individual variables could be cumbersome.
  - Why not create a data structure containing all the required parameters?

Acknowledgement: Photograph Abirami Selvaraj, Wikimedia Commons, May 2022.

What we already (should) know

In C, a data structure could be used.

- Define a data structure:-

```
typedef struct
{
   int   age;
   float weight;
} Cat_t;
```

This new type is `Cat_t`.

- We can now declare variables of type `Cat_t`

```
Cat_t   rufus;
Cat_t   felix;
```

- ...and populate their fields

```
rufus.age = 17;
rufus.weight = 3.4;

felix.weight = 2.7;
felix.age = 2;
```

Introduction to Classes

- Classes are collections of variables combined with related functions.
- Clients are functions or other classes that make use of your class.
- Encapsulation bundling together of information and capabilities into a single object. Result:
  - Clients only need to know what the class does.
  - Clients do not need to know how the class is implemented.
- Variables in a class are member variables or data members
- Functions in a class are member functions or methods of the class.

Declaring a Class

Example: A C++ analogy of the C `Cat_t` data structure

Class name

Opening Brace

```
Class Cat
{
public:
    int     age;
    float   weight;
    void Meow();
    void Purr();
};
```

Data Members

Member functions (methods)

Closing Brace, semicolon

This declaration does not allocate memory. It just tells the compiler how big the class is.

Defining an object

An object is an individual instance of a class.

Declared using following convention:-

```
Class_Name Object_name;
```

For example:-

```
Cat rufus;
Cat felix;
```

Compare this with what you know from C.

- Identical approach.

Accessing Class Members

Class members can be accessed using the dot (`.`) operator.

Example, accessing data members:-   (nothing new here!)

```
rufus.age = 17;
rufus.weight = 3.4;
```

and

```
felix.weight = 2.7;
felix.age = 2;
```

Example, accessing member functions:-

```
rufus.Purr();     // Calls the Purr() function for rufus
felix.Meow();     // Calls the Meow() function for felix.
```

Private vs Public?

- All members of the class are `private` by default.
- `private` members can only be accessed via class methods / functions.
- `public` members can be accessed through any object of the class.
- Accessing `private` data outside a class results in a compiler error.
- Good practice to make member data `private`.
  - Use accessor methods or accessor functions to set and retrieve values.

Implementing Member Functions

The member accessor functions are now also implemented using the following general approach:-

```
return_type class_name::function_name()
{
}
```

2 colons

For the two member functions, **`Meow()`** and **`Purr()`**

Class name

```
void Cat::Meow()
{
  playSoundFile("meow.wav");
}

void Cat::Purr()
{
  playSoundFile("purr.wav");
}
```

Very similar to implementing functions in C but with the additional **`class_name::`**

return type

Implementing Accessor Functions

Consider a slimmed-down declaration of the **Cat** class with a single `private` data member: **age**.

```
Class Cat
{
public:
   int  getAge();                  // Accessor function
                                   // to get age
   int  setAge(int updated_age);  // Accessor function
                                   // to set age
private:
   int  age;                       // age is now a private member
};
```

Implementing Accessor Functions

The member accessor functions are now also implemented using the following general approach:-

```
return_type class_name::function_name()
{
}
```

Two colons

For this case:-

Class name

```
int Cat::getAge()
{
   return age;
}

void Cat::setAge(int updated_age)
{
   age = updated_age;
}
```

return type

The `private` data member `age` can now be accessed using these `public` accessor functions

Using Accessor Functions

Example: creating a cat called 'bob', setting its age, reading its age and echoing it back…..

```
Cat bob;

void setup()
{
  Serial.begin(115200);
  bob.setAge(15);
  Serial.print("The age is ");
  Serial.println(bob.getAge());
}
```

Constructors & Destructors

Constructors:

- Member function without a return type - not even `void`
- Used for initializing member variables within a class.
- Constructor name = Name of Class

Destructors:

- Member function used for clean-up purposes
- Often empty
- Destructor name = ~Name of Class (note the tilde)
- Good practice to provide your own destructor even if the compiler implicitly creates one for you.

Constructors & Destructors

Defining a constructor and a destructor for the **`Cat`** class….

```
Class Cat
{
public:
  Cat(int ageOfCat);        // constructor
  Cat();                    // default constructor
  ~Cat();                   // destructor
     :
```

- A default constructor is created by the compiler if no constructor is declared. This does nothing.
- If a constructor is declared you are required to implement the default constructor.

Constructors & Destructors

Implementing constructors and a destructor for the `Cat` class: Adding to earlier example:-

```
Cat::Cat()                  // Default constructor
{                           // that does nothing.
}                           // e.g. Cat rufus;

Cat::Cat(int ageOfCat)      // Constructor called when a value of
{                           // age is passed as an argument.
  age = ageOfCat;           // e.g. Cat rufus(17);
}

Cat::~Cat()                 // Empty destructor
{
}
```

Good Practice

- Place class declarations in header `.h` files.
- Place class implementations in source `.cpp` files.
- Make member data `private` using accessor methods to access these (already covered).
- Incorporate constructors and destructors.

How Can Classes Facilitate Our Concurrency Model?

- The timing code for concurrent operation works well.
- It is likely to appear many times in a multi-modular piece of firmware.
- Why not encapsulate this in a Class?

How Can Classes Facilitate Our Concurrency Model?

- Let's declare a class called `Concurrent`

```
class Concurrent {
  public:
    bool              actionTask();
    void              setModuleDelay(unsigned long mod_delay);
    unsigned long     getModuleDelay();
    void              startRunning();
    void              stopRunning();
    Concurrent();

  private:
    unsigned long     module_time;    // based on current time
    unsigned long     module_delay;   // required time out
    bool              module_doStep;  // do we perform our task?
    bool              isRunning;      // is our module running or halted?
};
```

All the data members are `private`

Implementing Accessor Functions

The member functions (methods): `startRunning()` and `stopRunning()` respectively set the value of the variable `isRunning` to `true` or `false`.

The variable `isRunning` determines whether the code module is running and timing (`true`) or held in a stopped state (`false`).

```
void Concurrent::startRunning()
{
  isRunning = true;
}


void Concurrent::stopRunning()
{
  isRunning = false;
}
```

Implementing Accessor Functions

The member functions (methods): `setModuleDelay()` and `getModuleDelay()` respectively set or retrieve the value of the variable `module_delay`.

```
void Concurrent::setModuleDelay(unsigned long mod_delay)
{
  module_delay = mod_delay;
}


unsigned long Concurrent::getModuleDelay()
{
  return module_delay;
}
```

Implementing The Timing

The member function (method): `actionTask()` when called will perform the timing operation and return:-
`true` to indicate it is time to perform a task, or `false` if the module is held or if the timeout has not elapsed.
This must repeatedly be called within `loop()`.

```
bool Concurrent::actionTask()
{
  if (isRunning == false)
  {
    module_time = millis();                       // HALTED
    module_doStep = false;
  }
  else
  {
    unsigned long m = millis();                   // RUNNING
    if (((m - module_time)) > module_delay)
    {
       module_time = m;
       module_doStep = true;                      // TIME ELAPSED
    }
    else
    {
      module_doStep = false;                      // TIME NOT ELAPSED
    }
  }

  return module_doStep;
}
```

Finishing Touches

- Add the constructor:-

```
Concurrent::Concurrent()
{
  isRunning = false;
  module_delay = 1000;
  module_time = millis();
}
```

  - This puts the module in a sane state by setting the module delay to 1 second, setting the module time to the current time and by ensuring the module does not run
- The destructor is implicitly added by the compiler.
- The declarations are stored in the file Concurrent.h
- The implementations are stored in the file Concurrent.cpp

Putting it to use

Simply place Concurrent.h and Concurrent.cpp in your project folder. You may need to restart the Arduino IDE at this point.

Include the `Concurrent` class library:-

```
#include "Concurrent.h"
```

For each code module, create an instance of the `Concurrent` class.

- In this case we require three modules: one for the red LED, one for the yellow LED, one for the green LED. Name these instances appropriately.

```
Concurrent redControl;

Concurrent yellowControl;

Concurrent greenControl;
```

Putting it to use

In **`setup()`**, assign module delays to each object and set each object into a running state.

```
redControl.setModuleDelay(500);
redControl.startRunning();

yellowControl.setModuleDelay(300);
yellowControl.startRunning();

greenControl.setModuleDelay(600);
greenControl.startRunning();
```

Putting it to use

In `loop()`, for each object, see if the module's task needs actioning.

```
if (redControl.actionTask())
{
  redTask();
}

if (yellowControl.actionTask())
{
  yellowTask();
}

if (greenControl.actionTask())
{
  unsigned long new_delay;
  new_delay = greenTask();
  greenControl.setModuleDelay(new_delay);
}
```

Final Tidying

The task for the red and yellow LEDs can be packaged into functions.

```
void redTask()
{
  static int state = 0;
  switch (state)
  {
  case 0:
    HAL_ledRed1On;
    state = 1;
    break;
  case 1:
    HAL_ledRed1Off;
    state = 0;
    break;
  }
}
```

```
void yellowTask()
{
  static int state = 0;
  switch (state)
  {
  case 0:
    HAL_ledYellow1On;
    state = 1;
    break;
  case 1:
    HAL_ledYellow1Off;
    state = 0;
    break;
  }
}
```

Final Tidying

The task for the green LEDs can also be packaged into a function. This function returns the new required module delay; using this the module delay can be updated……

```
unsigned long greenTask()
{
  unsigned long new_time;
  static char state = 0;
  switch(state)
  {
  case 0:
    HAL_ledGreen1On;  // On for
    new_time = 600;   // 600ms
    state = 1;
    break;
  case 1:
    HAL_ledGreen1Off; // Off for
    new_time = 450;   // 450ms
    state = 2;
    break;
  case 2:
    HAL_ledGreen1On;  // On for
    new_time = 1800;  // 1800ms (1.8s)
    state = 3;
    break;
  case 3:
    HAL_ledGreen1Off; // Off for
    new_time = 1200;  // 1200ms (1.2s)
    state = 0;
    break;
  }
  return new_time;
}
```

Summary

- Examined the poor structure demonstrated in the first-pass at concurrency
  - Saw that just three modules led to convoluted code that is difficult to read, understand and maintain.
- Introduced the concept of classes.
- Used classes to encapsulate the timing.
  - This is the first step towards improving the structure of code capable of concurrency.
  - This is a massive improvement, but further improvements can be made……