Universidade Federal de Santa Catarina
Centro Tecnológico
Departamento de Automação e Sistemas
Engenharia de Controle e Automação
Introdução à Informática para Controle e Automação - DAS 5334

Aula 9 — Interface de Programação[1]

Nas aulas anteriores vimos como funções são úteis para, entre outras coisas, evitar repetição de código e simplificar a compreensão do programa (sua legibilidade). Uma preocupação que o desenvolvedor de funções deve ter é defini-las de forma que possam ser usadas em muitos programas, talvez até independente do problema específico sendo resolvido. Por exemplo, a função `analogRead()` é feita de tal forma que serve para qualquer aplicação do Arduino. Seu programador pensou muito bem no seu nome, parâmetros e valor devolvido. Ele pensou em como outros programadores irão utilizar essa função e outras a ela relacionadas. Ele pensou na *interface* de programação da função. O objetivo dessa aula é discutir um pouco o projeto dessas interfaces.

1 Encapsulamento

Encapsular é colocar um trecho de programa que tem um objetivo bem determinado em uma função, dando a esse trecho um nome. Por exemplo, analisando o programa abaixo (que é parte de uma solução para o Genius):

```
1  const int maxRounds  = 31; // Total de jogadas
2  int     roundCounter = -1; // contador de número de cores sorteadas
3  int     playCounter  = -1; // contador de jogadas do usuário
4  boolean gameOver     = false;
5
6  int randSeq[maxRounds]; // guarda a sequência de cores gerada
7
8  void loop() {
9    if (!gameOver) {
10
11     int lastBt = -1;
12     for (; lastBt == -1 ; ) {
13       for (int i = 0; i < 4; i++) {
14         if (digitalRead(buttons[i]) == HIGH) {
15           lastBt = i;
16         }
17       }
18     }
19
20     if (lastBt == randSeq[playCounter]) { // se acertou o botão
21       if (playCounter == maxRounds) {     // se é a última cor do round
22         ...                               //     vitória!
23         gameOver = true;
24       } else {
25         ...                               // acertou
26         if (playCounter == roundCounter) {  // se usuário chegou na última cor sorteada
27           ...                               //   sorteia a próxima
28         } else {
29           playCounter++;                    //   espera próxima jogada
30         }
31       }
32     } else {
33       ...   // perdeu!
34       gameOver = true;
```

[1]Baseado em conteúdo do livro Think Python 2nd Edition by Allen B. Downey e conteúdo da página oficial do Arduino.

```
35      }
36    }
37  }
```

percebemos que tentar entender a função `loop` é complicado! É difícil ter uma noção geral de como o programa funciona, mesmo tendo muitos detalhes omitidos por meio de `....` Entretanto, o programador sabe que o `loop` tem dois momentos: esperar o usuário apertar um botão (linhas 11 a 18) e então decidir o que fazer (linhas 20 a 36). A função `loop` na listagem abaixo é certamente mais fácil de ler, devido ao *encapsulamento* desses dois momentos em funções.

```
1  void loop() {
2    if (!gameOver) {
3      int lastBt = esperaBotao();
4      decide(lastBt);
5    }
6  }
7
8  int esperaBotao() {
9    for (; true ; ) { // essa linha poderia ser: while(true)
10     for (int i = 0; i < 4; i++) {
11       if (digitalRead(buttons[i]) == HIGH) {
12         return i;
13       }
14     }
15   }
16   return -1;
17 }
18
19 void decide(int lastBt) {
20   if (lastBt == randSeq[playCounter]) { // se acertou o b
21     .....
22   }
23 }
```

2 Generalização e Refinamento

No programa do Genius, em dois momentos se produz um som com luz associada: jogada certa e jogada errada. Com encapsulamento, podemos fazer duas rotinas para produzir o som/luz, como as que seguem:

```
1  const int errorTone  = 42;
2
3  void tocaAcertou(int bt) { // bt é o botão apertado pelo usuário
4    tone(buzzer, tones[bt]);
5
6    digitalWrite(lights[bt], HIGH);
7    delay(420);
8    digitalWrite(lights[bt], LOW);
9    noTone(buzzer);
10   delay(50);
11 }
12
13 void tocaErro(int bt) {
14   delay(80);
15   tone(buzzer, errorTone);
16
17   digitalWrite(lights[bt], HIGH);
18   delay(1500);
19   digitalWrite(lights[bt], LOW);
20   noTone(buzzer);
21   delay(20);
22 }
```

Podemos notar que a parte final das duas funções é muito parecida, mudando apenas o tempo dos delays. Podemos tirar essa parte comum e colocar em uma nova função — processo chamado de *refinamento*. Nessa nova função, os tempos de delay devem ser argumentos tornando a função *genérica* o suficiente para ser reutilizada nos dois casos:

```
1  void tocaAcertou(int bt) {
2    tone(buzzer, tones[bt]);
3    toca(bt, 420, 50);
4  }
5
6  void tocaErro(int bt) {
7    delay(80);
8    tone(buzzer, errorTone);
9    toca(bt, 1500, 20);
10 }
11
12 void toca(int led, int t1, int t2) {
13   digitalWrite(lights[led], HIGH);
14   delay(t1);
15   digitalWrite(lights[led], LOW);
16   noTone(buzzer);
17   delay(t2);
18 }
```

A nova função `toca()` é geral o suficiente para ser reaproveitada tanto em `tocaAcertou()` quanto `tocaErro()`. Em geral, quando trocamos valores absolutos (como 420 e 1500) por parâmetros (como t1 e t2) em uma função a tornamos mais geral.

O processo de refinamento pode continuar! A parte inicial das funções `tocaAcertou()` e `tocaErro()` não tem nem o mesmo número de linhas, mas é similar se imaginarmos um `delay(0)` no inicio de `tocaAcertou()`.

```
1  void tocaAcertou(int bt) {
2    toca(bt, tones[bt], 0, 420, 50);
3  }
4
5  void tocaErro(int bt) {
6    toca(bt, errorTone, 80, 1500, 20);
7  }
8
9  void toca(int led, int som, int antes, int durante, int depois) {
10   delay(antes);
11   tone(buzzer, som);
12   digitalWrite(lights[led], HIGH);
13   delay(durante);
14   digitalWrite(lights[led], LOW);
15   noTone(buzzer);
16   delay(depois);
17 }
```

3 Plano de desenvolvimento

A disciplina da quinta fase do curso Metodologia para Desenvolvimento de Sistemas (DAS 5312) aborda em detalhes métodos de modelagem e desenvolvimento de sistemas. Mas não precisamos aguardar até lá para executar boas práticas de programação. Com os conceitos vistos nesta aula, é possível seguir um plano de desenvolvimento de programas, isto é, um processo para a escrita de programas:

1. Escrever o programa sem funções.
2. Uma vez funcionando, identificar uma parte do código e encapsulá-la, dando um nome.
3. Generalizar a função com parâmetros.

4. Repetir os passos 1 a 3 até obter um conjunto de funções.

5. Procurar por oportunidades de melhorar o programa por meio de refinamento.

4 *NeoPixel*

NeoPixel é a marca para uma série de produtos fabricados pela Adafruit Industries nos quais LEDs RGB são endereçáveis por um único fio. São conjuntos de LEDs, individualmente chamados de pixels, combinados em diferentes formas como anéis, fitas, matrizes, etc. Cada LED pode assumir qualquer cor. Obviamente há vários produtos semelhantes no mercado de outros fabricantes e da própria Adafruit Industries. Porém, nos interessam aqueles da linha *NeoPixel*, em particular o *NeoPixel Jewel* e os *NeoPixel Rings* (anéis), por estarem disponíveis no simulador da Autodesk Circuits. Nesta aula, usaremos *NeoPixels* para trabalhar alguns conceitos importantes no desenvolvimento de um programa, especialmente na criação de funções.

Para ter uma ideia do potencial deste tipo de produto, assista ao vídeo Torre Central de Transmissão, SoulVision 2015 - TESTE I no YouTube. O sistema usado pelo sonoplasta desse evento musical foi desenvolvido por um Engenheiro de Controle e Automação formado pela UFSC. O sistema é escalável, flexível e configurável e isso só foi possível pois o engenheiro em questão empregou rigorosamente os conceitos desta aula.

O Autodesk Circuits oferece cinco componentes *NeoPixel* (Figura 1): o *NeoPixel* que contém apenas um LED RGB (um pixel), o *NeoPixel Jewel* com sete pixels, e três *NeoPixel Rings* com doze, dezesseis, e vinte e quatro pixels. Notar que esses componentes podem ser conectados em cascata para aumentar o número de pixels do conjunto.

Figura 1: Componentes *NeoPixel* disponíveis no Autodesk Circuits.

O restante desta seção é baseado no tutorial fornecido pela própria Adafruit Industries sobre o uso de *NeoPixels* com Arduino.

O código a seguir apresenta a estrutura básica de um programa para uso com *NeoPixels*:

```
#include <Adafruit_NeoPixel.h>

const int PIN = 6;

Adafruit_NeoPixel strip = Adafruit_NeoPixel(7, PIN, NEO_GRB + NEO_KHZ800);

void setup() {
  strip.begin();
  strip.show();
}

```

```
12  void loop() {
13
14  }
```

A linha 1 é necessária para termos acesso às funções do *NeoPixel*. Na linha 3 criamos uma constante inteira chamada `PIN` à qual foi atribuído o valor 6 referente ao pino ao qual o *NeoPixel* está conectado (poderia ser qualquer pino `PWM` da placa). Essa linha de código não é necessária, mas é conveniente caso queiramos mudar o pino do *NeoPixel*. A linha 5 permite operarmos o *NeoPixel* ligado à placa por meio do nome `strip` (poderíamos dar qualquer nome válido). Veja que a linha 5 contém a chamada à função `Adafruit_NeoPixel()` que recebe três argumentos. O primeiro argumento é o número de pixels a ser controlado, por exemplo, 7 no caso do *NeoPixel Jewel*. O segundo argumento é o pino ao qual o *NeoPixel* está conectado, nesse caso o pino 6, conforme a constante `PIN`. Omitimos os detalhes sobre o último argumento; o aluno interessado pode consultar a documentação do NeoPixel. O número de pixels definido durante a chamada de `Adafruit_NeoPixel()` pode ser recuperado com uma chamada à função `strip.numPixels()` que não recebe argumentos.

Na função `setup()`, linhas 8 e 9, são chamadas duas funções. A função `strip.begin()` prepara o Arduino para uso com o *NeoPixel*. A função `strip.show()` faz com que os pixels mostrem as cores para as quais foram configurados. Como nenhuma configuração foi feita, os pixels permanecem apagados. Ainda que não seja necessária, essa chamada a `strip.show()` é de interesse, pois garante que os pixels estarão apagados no início da operação ao invés de inciarem com uma configuração anterior. A seguir veremos funções que podem ser usadas para operar o *NeoPixel*.

As cores dos pixels são definidas segundo o sistema RGB. De maneira simplificada podemos dizer que o sistema RGB define uma cor pela soma das quantidades de vermelho (***R**ed*), verde (***G**reen*) e Azul (***B**lue*). Os valores RGB variam de 0 a 255 em que 0 corresponde à ausência daquela cor e 255 corresponde à quantidade máxima daquela cor. Assim, a combinação (R, G, B) = (0, 0, 0) corresponde à cor preta, e (R, G, B) = (255, 0, 0) corresponde à cor vermelha. Para definir a cor de um pixel é usada a função `strip.setPixelColor()`, que recebe quatro argumentos. O primeiro argumento é o número do pixel que deve ser configurado com uma dada cor. Esse número é um valor entre 0 e o número de pixels em `strip` menos 1. Ou seja, no caso de um *NeoPixel Jewel*, que contém sete pixels, os pixels são numerados de 0 a 6. Os demais parâmetros são os valores RGB de vermelho, verde e azul. Por exemplo, o comando `strip.setPixelColor(3, 148, 0, 211)` configura o quarto pixel com a cor violeta.

A função `strip.setPixelColor()` apenas configura o respectivo pixel. Para que a configuração entre em vigor, é necessário chamar a função `strip.show()`. Essa chamada pode ser feita após a configuração de apenas um pixel ou após a configuração de vários pixels.

Para algumas aplicações pode ser interessante usar a função `strip.setPixelColor()` com apenas dois argumentos. O primeiro argumento continua sendo o número do pixel a ser configurado e o segundo argumento é um único número que representa todas as informações RGB. Esse número pode ser gerado com a função `strip.Color()` que recebe três argumentos: os valores RGB de vermelho, verde e azul. Atenção para o fato de que o número gerado deve ser de um tipo inteiro que não vimos até agora, o `uint32_t`. Assim, se quisermos guardar a cor em uma variável, por exemplo a cor magenta, fazemos:

```
uint32_t magenta = strip.Color(255, 0, 255);
```

5 *Debugging*

Já deve estar claro que dividir um programa grande em funções menores melhora significativamente a legibilidade do programa. Outra consequência da divisão de programas em partes menores é a maior facilidade de realizar o *debugging*. Funções se tornam pontos naturais para verificação de erros. Resgatando o que vimos na seção de *debugging* da Aula 6, se uma função não está funcionado, há três possíveis causas:

- Há algo de errado com os argumentos passados para a função. Vimos que isso corresponde a uma violação das pré-condições. Neste caso, verificar os argumentos antes da chamada da função, seja com o *debugger* ou exibindo os valores num *display* é um bom caminho. Escrever códigos para verificar os argumentos também pode ser uma boa opção. Teste a função separadamente do resto do programa usando argumentos para os quais a resposta é conhecida. Cuidar porém, para não testar apenas uma alternativa que pode ser justamente a que não revela o problema.

- Há algo de errado com o corpo da função, isto é, ela não faz o que deveria fazer. Vimos que isso corresponde a uma violação das pós-condição. Verificar o valor devolvido ou mesmo valores intermediários de variáveis usadas na função é a melhor solução.
- Por fim, há algo de errado com a maneira como o valor devolvido é usado. Neste caso verifique o código e veja se o valor devolvido está sendo usado corretamente.

6 Problemas

A seguir resolveremos os problemas desta aula. Acesse o circuito da Aula 9 e faça uma cópia em sua própria conta com um clique no botão `Duplicate Project`. Esse circuito (Figura 2) possui um *Neopixel Jewel* conectado ao pino `D6`, um *NeoPixel Ring 16* conectado ao pino `D9` e três *Neopixels* (*Jewell*, *Ring 16*, e *Ring 24*) conectados em cascata ao pino `D10`. Um *display* também está conectado ao circuito como em diversas aulas anteriores.

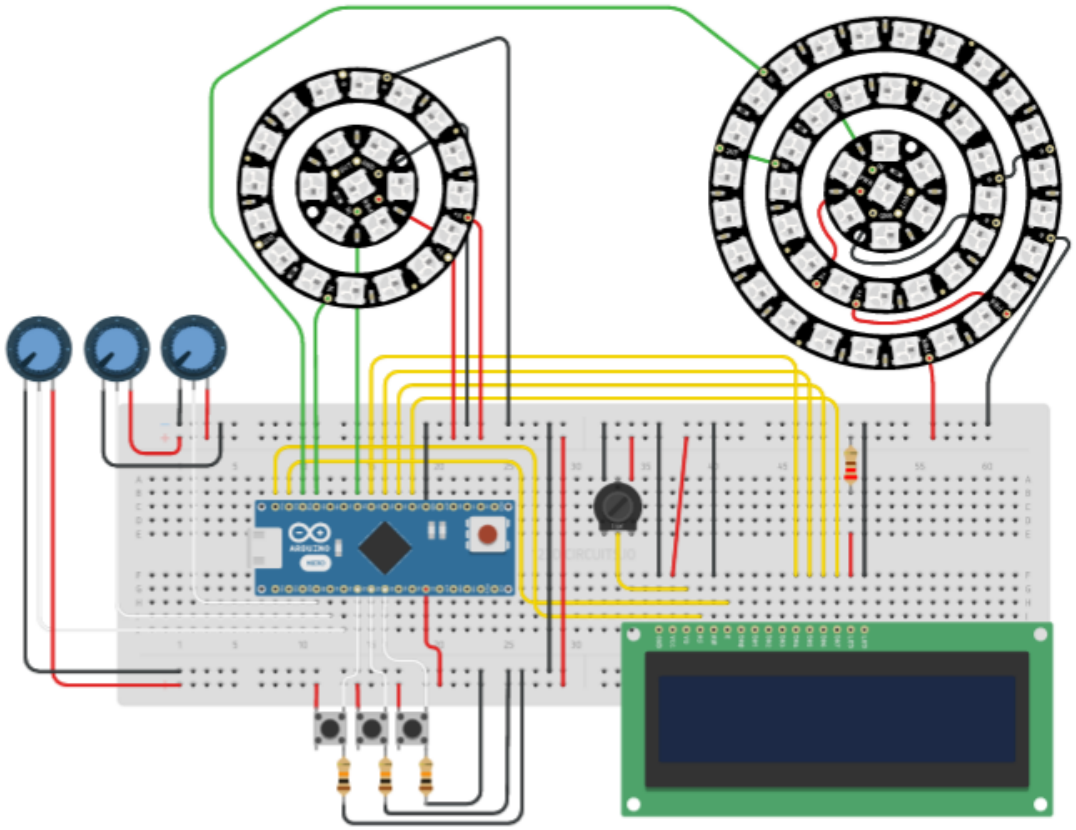

Figura 2: Circuito da Aula 9.

Problema 1

Escreva um programa que acenda em sequência com a cor vermelha cada um dos LEDs do NeoPixel Jewel conectado ao pino `D6` da placa a intervalos de 50 ms. Na sequência o programa deve mudar a cor da cada um dos LEDs para verde novamente em sequência a intervalos de 50 ms. O programa deverá ficar alternando as cores dos LEDs entre vermelho e verde sempre em sequência, isto é, fazendo cada vez uma varredura de todos os LEDs para uma cor e a seguir para a outra cor.

Problema 2

Na resolução do problema a repetição de código deve ter reduzido bastante, mas pode ser reduzida ainda mais. Use os conceitos de encapsulamento e generalização para escrever uma função `varreduraLEDs()`. Reflita bem sobre quais os parâmetros necessários.

Problema 3

Experimente usar a função `varreduraLEDs` com o NeoPixel Ring conectado ao pino `D9`. O que acontece? Use o conceito de generalização para a que a função funcione corretamente com qualquer número de LEDs. Mais um parâmetro é necessário?

Problema 4

Usando o ring de 24 LEDs, implemente um relógio. Use uma cor para indicar as horas, outra para indicar os minutos e outra para os segundos. A partir desta versão básica do relógio, acrescente as seguinte funcionalidades:

a) Use os potenciômetros para configurar a cor dos "ponteiros".

b) Use os botões para acertar a hora, implementar um cronometro e um alarme.

c) Use o ring de 7 LEDs indicar o dia da semana (domingo, segunda, ...).

d) Use o ring de 16 LEDs para indicar o número do dia no ano (dia 1, dia 2, ... até dia 365)! Pense em como representar esses 365 dias usando os 16 LEDs. Use combinações de LEDs, cores, ...