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 7 — *Arrays* e laços de repetição[1]

1 *Arrays*

Como vimos nas aulas anteriores, uma variável permite que um programa guarde na memória do computador um valor, o nome da variável é uma *referência* para um local na memória do computador. Alguns problemas precisam de programas que guardam na memória centenas ou milhares de valores e como as variáveis vistas até aqui fazem referência a um único valor, seria necessário criar centenas de variáveis! *Arrays* resolvem isso. Um *array* é um tipo especial de variável que pode referenciar vários valores. Dito de outra forma, um *array* é uma referência para um lugar de memória (ou seja, é uma variável) onde vários valores são armazenados. Assim como as variáveis, o *array* tem um nome e um tipo, mas tem também um tamanho, que indica quantos valores podem ser colocados na memória correspondente.

A declaração de um *array* é muito similar à declaração de variáveis como visto até aqui, simplesmente acrescenta-se o tamanho do *array*, entre colchetes, `[]`, após o nome. Por exemplo:

```
int v[5];
```

Nesse exemplo, foi criado um *array* com nome `v` para armazenar cinco valores inteiros. Cada um desses valores tem uma posição no *array*: a primeira posição é 0 e a última 4. Portanto, o primeiro valor desse *array* é referenciado por `v[0]`, o segundo por `v[1]`, ... e o último por `v[4]`. O valor entre colchetes, que indica a posição à qual estamos no referindo, é chamado de *indexador* ou índice do *array* e, muito importante, é uma expressão aritmética! Podemos escrever, por exemplo `v[1+3]`, `v[x/2]`, ...

Assim como uma variável, *arrays* podem ser declarados e inicializados no mesmo comando:

```
int v[5] = {4, 13, 8, 17, 21};
```

a posição 0 do *array* tem o valor 4, a posição 1 tem valor 13, ...

Para atribuir um valor para uma posição do *array*, utiliza-se o comando de atribuição com a variável do *array* indexada:

```
v[1] = 41;       // atribui 41 para a segunda posição do array
v[1] = v[1] + 1; // soma 1 à segunda posição do array
```

2 Operadores de incremento e decremento

Ao trabalhar com *arrays*, é usual termos que avançar ou voltar um ou mais elementos do *array*. Tipicamente isso é feito por meio de uma variável que corresponde ao índice ou posição de um elemento no *array* e que tem adicionado ou subtraído ao seu valor o número um, uma ou mais vezes. Para evitar a escrita de operações de adição, por exemplo `x = x + 1`, a linguagem Arduino oferece operadores compostos que simplificam a operação. Em geral o uso de operadores compostos não é recomendado, mais dois deles são de interesse. Os operadores de decremento (`--`) e incremento (`++`). São operadores unários (apenas um operando) que decrementam (diminuem) ou incrementam (aumentam) de um o valor do operando (que é uma variável). Por exemplo:

```
1 int x = 0;
2 x = x + 1;
```

é equivalente a

```
1 int x = 0;
2 ++x;
```

O caso do operador de decremento é análogo.

Os operador de decremento e incremento podem ser usados antes (prefixado) ou depois (posfixado) do operando. Assim, as duas formas a seguir são válidas:

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

```
x++;
++x;
```

No caso do exemplo, o resultado é o mesmo. Após a execução da operação, `x` terá aumentado seu valor em 1. Entretanto, é necessário ter cuidado no caso de seu uso em expressões. Por exemplo:

```
int x = 0;
int y;
y = ++x;
y = x++;
```

Ao final da linha 3, o valor de `y` e de `x` serão iguais a 1, como seria de se esperar, mas ao final da linha 4 o valor de `y` é 1 e o valor de `x` é 2. Isso acontece, pois quando o operador de decremento/incremento vem depois do operando, o valor referenciado pela variável e usado na avaliação da expressão é aquele de antes do incremento/decremento. Outra situação em que o uso do operador de decremento/incremento pode gerar dúvida é no caso de índices do *array*. Por exemplo:

```
int x = 0;
int y[5] = {1, 2, 3, 4, 5};
lcd.print(y[x++]);
lcd.print(y[++x]);
```

Nesse exemplo, a primeira chamada a `lcd.print` exibe no *display* o valor 1 ou o valor 2? E no segundo caso? Experimente. Nesta situação a dúvida reside em saber se o valor de `x` é devolvido antes ou depois de ocorrer o decremento/incremento. Construções desse tipo devem ser evitadas, pois deixam o código de difícil leitura e podem, inclusive, ser dependentes da arquitetura computacional utilizada.

3 Repetição simples

Não raro, comandos ou sequências de comandos precisam ser executados repetidas vezes em sequência. No caso de *arrays*, para podermos nos beneficiar das vantagens que oferece, o uso de repetição é indispensável. A repetição simples permite escrever apenas uma vez, mas executar várias vezes em sequência determinado comando ou sequência de comandos. Para isso usamos laços de repetição. O comando `for` da linguagem Arduino é um laço de repetição. A estrutura básica de um laço `for` é como segue:

```
for (<inicializacao>; <condicao>; <incremento>) {
  //comando(s);
}
```

Na linha 1 a palavra reservada `for` é seguida de parênteses, '`()`', com três conteúdos separados por ponto-e-vírgula, '`;`'. O primeiro conteúdo, que chamamos de inicialização é normalmente a declaração e a atribuição de valor a uma variável ou apenas a atribuição de um valor a uma variável já existente. Esse conteúdo é executado uma única vez no início da operação do laço de repetição. O segundo conteúdo é uma condição que deve ser verificada pelo laço antes de cada repetição. O laço só será executado *enquanto* essa condição for verdadeira. Por fim, o terceiro conteúdo é usualmente uma expressão de incremento ou decremento de uma variável que é executado ao final *de cada* repetição. Em geral as expressões do segundo e terceiro conteúdo usam a variável do primeiro, mas isso não é uma regra. Os parênteses são seguidos por um bloco de código, isto é, um ou mais comandos agrupados por chaves, '`{}`'. O código a seguir é um exemplo.

```
const int n = 5;
int numeros[n] = {4, 2, 3, 5, 1};
for (int i = 0; i < n; i++) {
  lcd.print(numeros[i]);
  lcd.home();
  delay(1000);
}
```

As duas primeiras linhas inicializam e declaram uma constante `n` do tipo inteiro com o tamanho do *array* e o próprio *array* inicializado com cinco valores. Atenção que o tamanho do *array*, se definido, deve ser feito por uma constante. Na linha 3, a variável `i` é declarada como um inteiro e tem atribuída à ela o valor `0`. Se a variável `i` já tivesse sido declarada anteriormente, seu tipo não deveria ser declarado novamente e a palavra reservada `int` seria omitida. A seguir a expressão `i < n` é verificada antes de cada repetição. Caso `i` seja igual ou maior que `n` o laço de repetição encerra. Por fim, a expressão `i++` incrementa de 1 o valor de `i` ao final

de cada repetição. Ainda na linha 3, inicia-se um bloco de comandos com uma chave de abertura. São três comandos nas linhas 4 a 6. O bloco de comandos é encerrado com o fechamento da chave na linha 7. A cada repetição executada pelo laço `for` esses três comandos serão executados. Notar como o valor da variável `i` é usado para a execução do laço e para acessar cada um dos elementos do *array* `numeros`.

Devemos ficar atentos para o fato que o funcionamento do Arduino é baseado em chamadas sucessivas à função `loop()`. Ou seja, o próprio Arduino funciona como se fosse um laço de repetição com "duração infinita". Assim, será necessário cuidado para escolher aquilo que deve ser escrito com o uso de um laço `for` ou que deve ser simplesmente repetido naturalmente por chamadas do Arduino à função `loop()`.

3.1 Padrão de travessia

A apresentação dos operadores de incremento e decremento na seção anterior sugere que ao usar *arrays* provavelmente teremos o interesse ou a necessidade de percorrer, ou atravessar, o *array*, passando por alguns ou todos os seus elementos. Esse tipo de computação segue um padrão bem definido e que se repetirá em diversas implementações. De fato, essa foi exatamente a estrutura do código na listagem da seção anterior. O padrão observado é o seguinte.

```
1 for (<limite inferior>; <variavel de contagem> < <limite superior>; <incremento>) {
2   <comandos>;
3 }
```

A travessia pode ocorrer no sentido contrário com a inversão dos limites, com a inversão do operador de comparação e com o uso do operador de decremento. A travessia também pode dar "passos" maiores do que 1, substituindo-se o operador de incremento/decremento por uma adição/subtração com passo desejado associada a um comando de atribuição.

3.2 Padrão de busca

Muitas vezes desejamos saber se um valor existe no *array*, e caso exista, podemos querer saber a posição do elemento no *array*. Mais uma vez, trata-se de uma computação que tem um padrão bem definido:

```
1 int indice = -1;
2 for (<limite inferior>; <variavel de contagem> < <limite superior>; <incremento>) {
3   if (<array>[<variavel de contagem>] == <valor desejado>) {
4     indice = <variavel de contagem>;
5   }
6 }
```

Notar que esse padrão é praticamente igual ao padrão de busca, mas inclui dentro do laço de repetição uma estrutura condicional. Nesse caso, a alteração do valor de `indice` acontecerá apenas se o valor encontrado no *array* for igual ao valor desejado. Notar, que se houver mais de um valor do *array* igual ao valor desejado, nessa implementação o índice do último encontrado ficará na variável `indice` depois da repetição. Se o valor não estiver no *array*, o `indice` permanecerá com o valor -1. Por exemplo:

```
1 const int n    = 5;
2 int numeros[n] = {4, 2, 3, 5, 1};
3 int valor_des  = 3;    // valor desejado
4 int indice     = -1;
5 for (int i = 0; i < n; i++) {
6   if (numeros[i] == valor_des) {
7     indice = i;
8   }
9 }
```

Alguém pode se perguntar se essa é a melhor maneira de fazer uma busca em um array. Certamente não é, mas isso é assunto para outra disciplina.

3.3 Contagem e acumulação

Laços de repetição também são muito úteis para atividades de contagem e acumulação, por exemplo, contar o número de ocorrências de um certo elemento no *array* ou acumular (somar) elementos específicos do array. A contagem é tipicamente realizada por meio dos operadores de incremento/decremento, enquanto a acumulação admite a soma de qualquer valor. O exemplo a seguir faz as duas tarefas com base em um *array*:

```
const int n          = 5;
int numeros[n] = {4, 2, 2, 5, 1};
int valor_des  = 2;
int acumulador = 0;
int contador   = 0;
for (int i = 0; i < n; i++) {
  if (numeros[i] <= valor_des) {
    contador++; // conta o número de ocorrências de valores
                // menores ou iguais ao valor desejado
    acumulador = acumulador + numeros[i]; // acumula os valores encontrados menores
                                          // ou iguais ao valor desejado;
  }
}
```

4 Exemplos

Para ilustrar os benefícios de *arrays* no desenvolvimento de programas, iremos reescrever alguns programas usados em aulas anteriores usando *arrays*.

O primeiro exemplo é o programa que mostra os números 17, 31 e 3 no *display*, visto na aula anterior. A solução com rotinas é suficiente para mostrar esses três números, mas se imaginarmos que o programa deve imprimir uma sequência maior de números, iremos perceber que *arrays* permitem uma solução muito mais elegante.

```
#include <LiquidCrystal.h>
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

void setup() {
  lcd.begin(16, 2);
}

// rotina para mostrar um número e esperar um pouco
void mostra(int n) {
  lcd.print(n);
  delay(500);
  lcd.clear();
}

const int tam = 10;   // quantidade de valores
int valores[tam] = { 17, 31, 3, 7, 31, 77, 99, 13, 29, 87 };

int i = 0;            // posição do número a ser mostrado

void loop() {
  mostra(valores[i]);
  i = ++i % tam;      // passa para o próximo número
}
```

O segundo exemplo é baseado no circuito Estacionamento 1 e também foi abordado na última aula (sobre definição de funções pelo programador). O programa que segue é uma solução possível para o Problema 2.

```
#include <LiquidCrystal.h>
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

void setup() {
  lcd.begin(16, 2);
  pinMode(0, OUTPUT);
  pinMode(1, OUTPUT);
  pinMode(6, OUTPUT);
  pinMode(8, OUTPUT);
  pinMode(10, OUTPUT);
  pinMode(A3, INPUT);
```

```
  pinMode(A2, INPUT);
  pinMode(A1, INPUT);
  pinMode(A0, INPUT);
}

int somaLigados() {
    return digitalRead(A3) + digitalRead(A2) + digitalRead(A1) + digitalRead(A0);
}

boolean lotacao(int ligados, int nroPessoas) {
    return ligados >= nroPessoas;
}

void loop() {
  int ligados = somaLigados();
  digitalWrite( 0, !lotacao(ligados, 1));
  digitalWrite( 1,  lotacao(ligados, 1));

  digitalWrite( 6, lotacao(ligados, 2));
  digitalWrite( 8, lotacao(ligados, 2));
  digitalWrite(10, lotacao(ligados, 2));

  delay(500);
  lcd.print(ligados);
  lcd.home();
}
```

Podemos perceber que na rotina `loop` três vezes se repete o código `digitalWrite( p, lotacao(ligados, 2));`, onde `p` tem valores diferentes em cada execução do `digitalWrite`: 6, 8 e 10. Se colocarmos esses valores em um *array*, podemos usar uma repetição `for` para simplificar o programa. O mesmo acontece com as portas de entrada. Segue um versão do programa usando *arrays*.

```
#include <LiquidCrystal.h>
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

const int nroOuts = 5;     // quantidade de portas out e in
const int nroIns  = 4;

int portasOut[]   = {0,1,6,8,10};   // valores das portas
int portasIn[]    = {A3,A2,A1,A0};

void setup() {
  lcd.begin(16, 2);

  for (int i=0; i < nroOuts; i++)   // para todas as portas no array:
    pinMode(portasOut[i], OUTPUT);  //   define a porta como saída
  for (int i=0; i < nroIns; i++)    // idem para portas de entrada
    pinMode(portasIn[i], INPUT);
}

int somaLigados() {
  // usa o padrão de acumulador com repetição para somar as portas de entrada
  int a = 0;
  for (int i=0; i < nroIns; i++)
    a = a + digitalRead(portasIn[i]);
  return a;
}

boolean lotacao(int ligados, int nroPessoas) {
    return ligados >= nroPessoas;
}
```

```
void loop() {
  int ligados = somaLigados();

  digitalWrite( 0, !lotacao(ligados, 1));
  digitalWrite( 1,  lotacao(ligados, 1));

  for (int i=2; i < nroOuts; i++)   // para as portas de saída a partir da terceira
    digitalWrite(portasOut[i], lotacao(ligados, 2));

  delay(500);
  lcd.print(ligados);
  lcd.home();
}
```

Nesta versão do programa é fácil de incluir ou excluir portas tanto de entrada como de saída, as únicas linhas que precisam ser alteradas são as linhas com a declaração dos *arrays*.

5 *Debugging*

Como visto nas seções anteriores, o uso de *arrays* depende da manipulação de índices tanto para acessar os valores do *array* como para alterá-los. Você deve ter notado nas listagens apresentadas até aqui que, invariavelmente, uma variável com o tamanho do *array* acompanha o próprio *array* nas implementações. Isto ocorre, pois até o momento não temos recursos para descobrir o tamanho de um *array*. Conhecer o tamanho do *array* e a forma de indexação, que vai de zero até o tamanho do *array* menos um, garante que não tentaremos alterar ou acessar uma posição fora do array. Os exemplo a seguir mostra os dois casos:

```
1  int array[] = {5, 7, 2, 4, 2, 8}; // Array com 6 elementos, índice de 0 a 5
2  lcd.print(array[8]);              // Tenta mostra no display um valor em uma
3                                    // a posição maior que o tamanho do array
4  array[7] = 9;                     // Tenta atribuir o valor 9 em uma
5                                    // posição maior que o tamanho do array
```

Nos dois casos os resultados podem ser imprevisíveis. Tipicamente no caso linha 2 o valor lido será algum "lixo", um valor qualquer. No caso da linha 4, o valor de outra variável pode ser alterado ou o funcionamento completo do programa interrompido ou prejudicado. Resumidamente, tenha sempre cuidado com o tamanho do *array* ao usar índices.

6 Problemas

Problema 1

Releia o enunciado do Problema 4 da Aula 3. Escreva um programa semelhante para o circuito da Aula 2, mas que use *arrays* para fazer os LEDs piscarem em sequência. Todos os 8 LEDs devem ser usados e LEDs verdes e vermelhos devem piscar alternadamente, isto é, primeiro um verde, depois um vermelho, depois outro verde, depois outro vermelho, e assim por diante. A velocidade com que eles piscam deve variar conforme o número de chaves acionadas conforme descrito no Problema 4 da Aula 3. O acionamento incorreto das chaves deve resultar em uma mensagem de erro.

Problema 2

Considerando novamente o Estacionamento 2 da Aula 4 (sobre vagas livres em um estacionamento) e a implementação feita na última aula (usando funções), reescreva o programa usando *arrays* procurando generalizar para qualquer número de vagas e não somente quatro. Utilize o programa abaixo como ponto de partida.

```
#include <LiquidCrystal.h>
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

// variáveis para guardar o estado do estacionamento
bool vagaLivreA0     = true;
int  ultA0           = LOW;

bool vagaLivreA1     = true;
int  ultA1           = LOW;

bool vagaLivreA2     = true;
int  ultA2           = LOW;

bool vagaLivreA3     = true;
int  ultA3           = LOW;

void setup() {
  lcd.begin(16, 2);

  pinMode(A0, INPUT);
  pinMode(10, OUTPUT);
  pinMode(13, OUTPUT);

  pinMode(A1, INPUT);
  pinMode(8, OUTPUT);
  pinMode(9, OUTPUT);

  pinMode(A2, INPUT);
  pinMode(6, OUTPUT);
  pinMode(7, OUTPUT);

  pinMode(A3, INPUT);
  pinMode(1, OUTPUT);
  pinMode(0, OUTPUT);
}

bool verificaVaga(int ultValor, int porta, bool estado, int ledVerde, int ledVermelho) {
  if (ultValor == LOW && ultValor != digitalRead(porta)) {
    // mudou de LOW para HIGH, carro entrando ou saindo
    estado = !estado; // muda o estado da variável
  }
  digitalWrite(ledVerde, estado);
  digitalWrite(ledVermelho, !estado);
  return estado; // retorna o novo estado da variável
}

void loop() {
  // controle da vaga A0
  vagaLivreA0 = verificaVaga(ultA0, A0, vagaLivreA0, 10, 13);
  ultA0 = digitalRead(A0); // não dá pra colocar dentro da função (com o que sabemos até agora)

  // vaga A1 (mesma idéia de programa que acima)
  vagaLivreA1 = verificaVaga(ultA1, A1, vagaLivreA1, 8, 9);
  ultA1 = digitalRead(A1);

  // vaga A2
  vagaLivreA2 = verificaVaga(ultA2, A2, vagaLivreA2, 6, 7);
```

```
  ultA2 = digitalRead(A2);

  // vaga A3
  vagaLivreA3 = verificaVaga(ultA3, A3, vagaLivreA3, 1, 0);
  ultA3 = digitalRead(A3);

  lcd.home();
  lcd.print(vagaLivreA3+vagaLivreA2);
  lcd.print(" <- vagas -> ");
  lcd.print(vagaLivreA1+vagaLivreA0);
}
```