Capítulo 6

Introdução à programação do Arduino

Assim como o uso de uma linguagem não está limitada apenas a livros e revistas, as linguagens de programação não estão estritamente relacionadas com algum tipo especial de microcontrolador, processador, computador, ou sistema operacional. A linguagem C é uma linguagem de programação considerada de alto nível e de uso geral. Entretanto, o fato da não associatividade da linguagem C a nenhuma máquina pode causar problemas durante a sua utilização, dependendo das peculiaridades de cada máquina na qual está se programando.

6.1 Introdução

Um programa é composto de uma sequência de comandos, normalmente escritos em um arquivo de texto. A ideia associada a escrever um programa para um microcontrolador é quebrar um grande problema em vários problemas menores (dividir para conquistar), mais simples de serem resolvidos. Suponha, por exemplo, que é necessário escrever um programa para realizar a medição de temperatura e mostrar o resultado em um display. O processo de medição é feito com o uso de um sensor apropriado, que converte a temperatura em um valor proporcional de tensão analógica, por exemplo. O microcontrolador usa seu conversor A/D para converter essa tensão (analógica) em um número (digital) que é então enviado para o LCD através de um barramento. Assim, o programa para executar essa tarefa pode ser dividido em quatro partes que precisam ser executadas na ordem correta, como seguem:

1. Ativar e configurar o conversor A/D;
2. Realizar a medição do valor analógico;
3. Calcular a temperatura com base na relação existente entre a tensão analógica e a grandeza física;
4. Enviar de forma apropriada o valor calculado para o display.

Para facilitar a compreensão da linguagem de programação do Arduino, que é baseada na linguagem C/C++, vejamos primeiramente um exemplo de programa, que no ambiente de desenvolvimento Arduino é chamado de “sketch”:

```
//Primeiro programa - sketch

void setup()
{
  pinMode(13, OUTPUT);
}

void loop()
{
  digitalWrite(13, HIGH);
  delay(1000);
  digitalWrite(13, LOW);
  delay(1000);
}
```

Vejamos o significado de cada linha do programa. A primeira linha do programa é chamada de comentário:

```
//Primeiro programa - sketch
```

Os comentários não são interpretados pelo compilador, isto é, eles não geram código executável para o microcontrolador, sendo apenas descrições inseridas no código-fonte pelo programador com o intuito de documentar o programa e para facilitar o seu entendimento por parte de outros programadores que eventualmente farão alterações ou atualizações no programa original.

Os comentários podem ser de linha simples, como foi mostrado no exemplo, e são iniciados por uma barra dupla. Os comentários de linha simples podem ser iniciados em qualquer ponto de uma linha e são muito utilizados para descrever o funcionamento ao final de cada linha de código. Entretanto, após iniciado um comentário de linha simples, todo o restante da linha será tratado como comentário.

Os comentários também podem ser de múltiplas linhas. Nesse caso, são compostos por uma ou mais linhas. Os caracteres "/*" iniciam o comentário e a sequência "*/" terminam o comentário, como mostra o exemplo abaixo:

```
/*
Este é um exemplo
de comentário de
múltiplas linhas
*/
```

Na linha seguinte do programa temos:

```
void setup()
```

A declaração `void setup()` especifica o nome de uma função (*setup*). Uma função é um conjunto de instruções que pode ser executado a partir de qualquer ponto do programa. O sinal de abertura da chave "{" é utilizado para delimitar o início da função e o sinal de fechamento da chave "}" indica o final da função. Na realidade, as chaves delimitam o que chamamos de bloco de programa ou bloco de código.

A função setup( ) é uma função padronizada da linguagem Arduino, devendo estar contida em qualquer programa. Normalmente, o bloco de código contido dentro dessa função executa instruções relativas às configurações de inicialização do programa e dos circuitos internos do microcontrolador que serão usados na aplicação. Alguns exemplos de configurações são: Configurar um pino como entrada ou saída digital, ativar o conversor analógico/digital, configurar o limite de contagem de um contador, configurar as interrupções, etc.

No exemplo, o bloco de programa dentro da função setup( ) é formado por uma única instrução. Cada instrução é terminada por um ponto e vírgula ";". A instrução `pinMode(13, OUTPUT)` é uma chamada a uma função interna da linguagem. A instrução configura o pino 13 da placa como um pino de saída digital (OUTPUT). Pinos configurados como saída digital podem ser posteriormente setados (levados ao nível lógico alto, fazendo com que uma tensão de 5 V apareça no pino) ou resetados (levados ao nível lógico baixo, fazendo com que uma tensão de 0 V apareça no pino) por meio de outras instruções.

A instrução pinMode( ) pode ser utilizada em qualquer um dos pinos do Arduino, inclusive nos pinos de entrada analógica (o que faz com que as entradas analógicas passem a operar como pinos digitais). Dizemos que a instrução requer dois parâmetros: o número do pino e o modo de funcionamento (entrada/saída). Os parâmetros de uma instrução ou função é o conteúdo que aparece dentro dos parênteses, como em `pinMode(13, OUTPUT)`. Nas funções, os parâmetros também são chamados de "argumentos".

A declaração `void loop()` especifica o nome de outra função padronizada da linguagem. A função loop( ), por sua vez, é formada por 4 instruções:

* A instrução `digitalWrite(13, HIGH)` leva o pino 13, inicialmente configurado como saída digital na função setup(), para o estado lógico alto (HIGH), enviando 5 V a este pino;
* A instrução `delay(1000)` gera um atraso na execução do programa de 1000 ms (1 segundo);
* A instrução `digitalWrite(13, LOW)` leva o pino 13 para o estado lógico baixo, enviando 0 V a este pino;
* Mais uma vez, a instrução `delay(1000)` pausa a execução do programa durante 1000 ms.

Aqui cabe um esclarecimento sobre como as funções setup( ) e loop( ) são tratadas no Arduino: Quando o Arduino é ligado, a função setup( ) é inicialmente chamada, e o bloco de código equivalente é executado uma única vez. A partir de então, a função loop( ) fica sendo chamada repetidamente até que o Arduino seja desligado ou reiniciado, e o bloco de código é executada continuamente.

Dessa forma, o objetivo do programa exemplo é: Após a função setup( ) configurar o pino 13 como saída, e sabendo que há um LED conectado a este pino, durante a repetição da função loop( ), este LED será aceso, o programa aguardará 1 segundo, mantendo-o aceso, e em seguida o LED será apagado, mantendo-o apagado por mais um período de um segundo. O LED ficará piscando então a cada 1 segundo. Obviamente este é um programa bem simples, mas é útil para o entendimento inicial da estrutura de um programa.

Podemos dizer que um programa é constituído por um ou mais dos seguintes elementos:

*__Operadores__ - São elementos utilizados para comandar interações entre variáveis e dados;

*__Variáveis__ - São usadas para armazenamento temporário ou permanente de dados. O Arduino dispõe de uma variedade de tipos de variáveis e dados;

*__Estrutura de controle__ - São elementos essenciais à escrita de programas. São utilizados para controlar, testar e manipular dados e informações dentro de programas;

*__Funções__ - São estruturas de programa utilizadas para simplificar, otimizar ou apenas tornar mais claro o funcionamento do programa.

O programa abaixo executa a mesma tarefa do programa anterior, com exceção do tempo em que o pino 13 permanece em nível lógico alto, mas foi escrito de forma diferente para abordarmos outras características da linguagem.

```
//Segundo programa

int tempo=1000;

void setup()
{
  pinMode(13, OUTPUT);
}

void loop()
{
  digitalWrite(13, HIGH);
  delay(tempo);
  digitalWrite(13, LOW);
  delay(tempo);
}
```

Nesse exemplo, o primeiro comando `int tempo=1000` é chamado de declaração de variável e atribuição de valor a uma variável. Este comando determina que o compilador crie uma variável do tipo inteira, **int**, chamada **tempo**. Esta operação, na realidade irá reservar um espaço de memória no microcontrolador para o armazenamento do valor relativo à variável. O tipo **int** especifica um tipo de dado de 16 bits com valores decimais compreendidos entre -32768 e 32767. Veremos mais sobre variáveis e tipos de dados em seguida.

O nome dado à variável é um **identificador** e pode ser composto de letras, números e o caractere sublinhado "_". Veremos mais sobre identificadores válidos.

Na mesma linha, o termo `tempo=1000` constitui uma operação de atribuição. As atribuições são executadas pelo operador de igualdade "=". Desta forma, o termo `tempo=1000` fará com que o compilador gere uma sequência de instruções para fazer com que o valor 1000, em decimal, seja armazenado na variável **tempo**.

A outra diferença que vemos nesse segundo exemplo são as linhas com a instrução `delay(tempo)`. Como vimos, esta também é uma função interna do compilador e é utilizada para gerar um atraso de alguns milissegundos na execução do programa. O parâmetro dessa função é igual ao conteúdo da variável tempo, que foi previamente carregada com o valor 1000. Logo, o atraso gerado será de 1000 ms.

A vantagem de usarmos uma variável como parâmetro da função delay( ) em comparação com um valor fixo, como foi o caso mostrado no primeiro exemplo, é que ao alterarmos o valor da variável na primeira linha do programa o atraso será alterado nas duas chamadas ao comando delay( ) dentro da função loop. Essa estratégia também pode ser usada para alterar dinamicamente o valor do atraso durante a execução do programa.

Quando o programador altera o valor de uma variável durante a etapa de programação, dizemos que a mudança ocorre em tempo de compilação. Já quando a variável é alterada durante a execução do programa, por uma instrução ou qualquer outra interação externa, dizemos que a mudança ocorre dinamicamente em tempo de execução.

6.2 Palavras reservadas da linguagem Arduino

Toda linguagem de programação, incluindo a linguagem Arduino, possui um conjunto de palavras ou comandos para os quais já existe interpretação interna prévia. Tais palavras não podem ser utilizadas para outras finalidades que não as definidas pela linguagem.

A interface do IDE do Arduino tem uma caraterística chamada de "realce de sintaxe", que faz com que algumas palavras digitadas mudem de cor, indicando se tratar de uma palavra reservada da linguagem. Por exemplo, ao digitarmos as palavras setup, loop, digitalWrite, delay, OUTPUT, HIGH e LOW, essas se apresentam com cores distintas do restante do texto.

6.3 Identificadores

Identificadores são nomes dados pelo programador a variáveis, funções e outros elementos da linguagem. Não é permitido utilizar palavras reservadas como identificadores.

Um identificador pode ser composto de caracteres numéricos e alfanuméricos, além do caractere sublinhado "_". Além disso, um identificador somente pode ser iniciado por uma letra ou sublinhado, nunca por um número, como nos exemplos:

```
variável
Variavel1
_teste
_Teste1
_13_abc
abc_DEF
```

A linguagem Arduino é do tipo "*case sensitive*". Isso significa dizer que ela faz distinção entre letras maiúsculas e minúsculas nos identificadores.

6.4 Variáveis e tipos de dados

Variáveis e constantes são os elementos básicos que um programa manipula. Uma variável é um espaço reservado na memória do microcontrolador para armazenar um tipo de dado determinado. Variáveis devem receber nomes para poderem ser referenciadas e modificadas quando necessário. Muitas linguagens de programação exigem que os programas contenham declarações que especifiquem de que tipo são as variáveis que ele utilizará e às vezes um valor inicial. Tipos podem ser, por exemplo: inteiros, reais, caracteres, etc.

Os dados podem assumir os seguintes tipos básicos na linguagem Arduino:

Tipo	Tamanho em bits (bytes)	Intervalo
char	8 (1)	-128 a 127
int	16 (2)	-32768 a 32767
float	32 (4)	$-3.4*10^{38}$ a $3.4*10^{38}$
void	0 (0)	Nenhum valor

char: (Caractere) O valor armazenado é um caractere (letras, dígitos e símbolos especiais). Caracteres geralmente são armazenados em códigos (usualmente o código ASCII).
int: Número inteiro é o tipo padrão e o seu tamanho normalmente depende da máquina em que o programa está rodando. No caso do Arduino, seu tamanho é de 16 bits.
float: Número em ponto flutuante de precisão simples. São conhecidos normalmente como números reais.
void: Este tipo serve para indicar que um resultado não tem um tipo definido.

O intervalo especifica qual a faixa de valores decimais que podem ser representados por uma variável de um determinado tipo.

Modificadores podem ser aplicados aos tipos básicos. Estes modificadores são palavras que alteram o tamanho do intervalo de valores que o tipo pode representar. Por exemplo, um modificador permite que possam ser armazenados números inteiros maiores. Outro modificador obriga que só números sem sinal possam ser armazenados pela variável. Deste modo não é necessário guardar o bit de sinal do número e somente números positivos são armazenados. O resultado prático é que o conjunto praticamente dobra de tamanho.

A tabela abaixo mostra todos os tipos básicos definidos no Arduino:

Tipo	Tamanho em bits (bytes)	Intervalo
boolean	8 (1)	TRUE ou FALSE
char	8 (1)	-128 a 127
unsigned char, byte	8 (1)	0 a 255
int, short	16 (2)	-32768 a 32767
unsigned int, word	16 (2)	0 a 65536
float, double	32 (4)	$-3.4*10^{38}$ a $3.4*10^{38}$
long	32 (4)	-2147483648 a 2147483647
unsigned long	32 (4)	0 a $2^{32}-1$

6.5 Declaração de variáveis

Antes de serem usadas, as variáveis precisam ser declaradas para que o compilador possa reservar espaço na memória para o valor a ser armazenado. Declarar uma variável nada mais é do que informar ao compilador o nome dessa variável, através de um identificador, e o seu tipo, conforme a tabela anterior.

A forma geral de uma declaração é:

```
tipo lista_de_variaveis;
```

Exemplos:

```
int i;
unsigned int a, b, c;
unsigned long dia, mes, ano;
float salario;
```

Outro aspecto importante da declaração de variáveis é o local onde elas são declaradas. A importância do local onde a variável é declarada relaciona-se à acessibilidade ou não dessa variável por outras partes do programa.

Basicamente, podemos declarar as variáveis em dois pontos distintos do programa, a saber:

* No corpo principal do programa, (fora de qualquer função, inclusive das funções setup( ) e loop( )). Variáveis declaradas dessa forma são chamadas de variáveis globais, porque podem ser acessadas de qualquer ponto do programa. Dizemos que a variável tem um “escopo global”.

* Dentro de uma função. As variáveis declaradas dentro de uma função somente podem ser acessadas de dentro da função em que foram declaradas. Variáveis declaradas dessa forma são chamadas de variáveis locais. Isto significa que uma variável local somente existe enquanto a função está sendo executada. Dizemos que a variável tem um “escopo local”.

Em seguida, temos um exemplo de um programa com variáveis locais e globais.

```
//Terceiro programa (variáveis locais e globais)

int var_global;

void setup()
{
  pinMode(13, OUTPUT);
}

void loop()
{
  int tempo, var_local;
  var_global=3;
  var_local=5;
}
```

Apesar do programa anterior não produzir nenhum efeito sobre os pinos do microcontrolador, ilustra a forma de declaração de variáveis globais e locais.

Fora das funções setup( ) e loop( ), temos a declaração de uma variável do tipo **inteiro** cujo nome atribuído foi **var_global**.

Dentro da função loop( ) temos a declaração de duas variáveis do tipo inteiras, cujos nomes são **tempo** e **var_local**. Essas variáveis são locais, pois foram declaradas dentro do corpo dessa função. As variáveis locais devem sempre ser declaradas no início de cada função, antes do chamado a qualquer comando.

No final da função loop( ) as variáveis local e global são carregadas pela atribuição do valor 3 à variável global e do valor 5 à variável local.

6.6 Operadores

Operadores são elementos da linguagem que realizam ações sobre variáveis. Podemos classificar os operadores nas categorias principais que seguem abaixo:

Operador de atribuição (igualdade): comum à maioria das linguagens, este operador (=) faz a variável da esquerda assumir o valor da variável, constante ou expressão da direita. Exemplo:
`var = var + 2;` (Portanto, o conteúdo de var é aumentado de 2).

Operadores aritméticos: realizam operações aritméticas sobre os valores das variáveis, constantes ou expressões associadas.

Operador	Descrição
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Resto da divisão

Exemplo:
`var = 10 % 3;` (o valor de var será 1).

Operadores relacionais: Eles avaliam o relacionamento entre duas expressões e dão o resultado 1 (VERDADEIRO), se a avaliação for verdadeira, ou 0 (FALSO) se for falsa.

Operador	Descrição
<	Menor que
<=	Menor ou igual
>	Maior que
>=	Maior ou igual
==	Igual
!=	Diferente

Exemplo:

```
var = 2;
if( var>2 )
 var = 3;
```
(Ou seja, o valor de *var* continua sendo 2).

Um lapso comum é confundir o operador de comparação (==) com o operador de igualdade (=). Exemplo:

```
var = 1;
if( var=2 )
 var = 3;
```

O valor de var será 3, o que certamente não era esperado. Isso ocorre porque var=2 atribui 2 a var e, quando for executada, a expressão retornará um valor verdadeiro para a declaração if, permitindo a execução da linha seguinte. Portanto, o correto seria:

```
if( var==2 )
```

Deve ser enfatizado que, mesmo sem operadores, a linguagem Arduino considera verdadeiro qualquer valor não nulo. Por exemplo, após a execução das linhas

```
var = 2;
if( -50 )
var = 3;
```
(o valor de var será 3).

Operadores de incremento e decremento: São dois operadores bastante úteis para simplificar expressões:

```
++ (incremento de 1)
-- (decremento de 1)
```

Podem ser colocados antes ou depois da variável a modificar. Se inseridos antes, modificam o valor antes de a expressão ser usada e, se inseridos depois, modificam depois do uso. Alguns exemplos:

```
x = 2;
var = ++x;
```

No caso acima, o valor de var será 3 e o de x será 3.

Operadores de bits: Manipulam bits em valores inteiros. Nos exemplos a seguir, são considerados dados de 8 bits e sem sinal.

~ *(complemento)*: Atuando em apenas um valor, muda os bits de valor 1 para 0 e vice-versa. Exemplo: se a variável var tem o valor 170 (10101010), após `~var`, ela terá 85 (01010101).

<< (deslocamento à esquerda): Desloca para a esquerda os bits do operando esquerdo no valor dado pelo operando direito. Equivale à multiplicação pela potência de 2 dada por este último. Exemplo: se a variável var tem o valor 3 (00000011), após `var << 2`, ela será 12 (00001100).

>> (deslocamento à direita): Desloca para a direita os bits do operando esquerdo no valor dado pelo operando direito. Equivale à divisão pela potência de 2 dada por este último. Exemplo: se a variável var tem o valor 12 (00001100), após `var >> 2`, terá 3 (00000011).

& (E): Faz o valor do bit igual a 1 se ambos os bits correspondentes nos operandos são 1 e 0 nos demais casos. Exemplo: se a variável var tem o valor 12 (00001100), fazendo essa operação com 6 (00000110), o resultado, `var & 6`, será 4 (00000100).

| *(OU):* Faz o valor do bit igual a 1 se um ou ambos os bits correspondentes nos operandos é 1 e 0 nos demais casos. Exemplo: se a variável var tem o valor 12 (00001100) e fazendo a operação com 6 (00000110), o resultado, `var | 6`, será 14 (00001110).

^ *(OU exclusivo):* Faz o valor do bit igual a 1 se apenas um dos bits correspondentes nos operandos é 1 e 0 nos demais casos. Exemplo: se a variável var tem o valor 12 (00001100) e fazendo a operação com 6 (00000110), o resultado, `var ^ 6`, será 10 (00001010).

Usar um desses operadores só altera as variáveis alvo se houver um comando de atribuição associado à operação.

Operadores lógicos: Usados normalmente com expressões booleanas, isto é, expressões que retornam verdadeiro ou falso (1 ou 0), para fins de testes em declarações condicionais.

&& (E lógico): Retorna verdadeiro se ambos os operandos são verdadeiros e falso nos demais casos. Exemplo: `if( a>3 && b<4).`

|| *(OU lógico)*: Retorna verdadeiro se um ou ambos os operandos são verdadeiros e falso se ambos são falsos. Exemplo:
`if( a>3 || b<4).`

! (NÃO lógico): Usado com apenas um operando. Retorna verdadeiro se ele é falso e vice-versa. Exemplo:
`if( !var )` Notar que essa expressão é equivalente a `if( var ==0 ).`

6.7 Declarações de controle

As declarações ou comandos de controle são uma parte muito importante de uma linguagem de programação. Podemos classificar as declarações de controle em duas categorias básicas:

1. Declarações de teste condicional: são utilizadas para testar determinadas condições/variáveis e executar um bloco de código para cada caso. A linguagem Arduino dispõe de dois tipos de declarações condicionais: a estrutura **if-else** e a estrutura **switch-case**;
2. Declarações de estrutura de repetição: são utilizadas para provocar a execução de um bloco de comandos enquanto uma determinada condição for verdadeira. A linguagem dispõe de três declarações de repetição: **while**, **do-while** e **for**.

O laço while( )

O laço **while** é um comando de repetição que possui a seguinte forma geral:

```
while(condição)
{
   comando1;
   comando2;
   ...
}
```

A filosofia de funcionamento do laço **while** é: primeiramente a condição é avaliada: Caso seja verdadeira, então o comando ou o bloco de comandos associado é executado e a condição é novamente avaliada, reiniciando o laço. Caso a condição seja falsa, o comando ou o bloco de comandos não é executado e o programa tem sequência a partir da declaração seguinte ao laço **while**. A condição testada pelo laço **while** pode ser qualquer expressão da linguagem Arduino que possa ser avaliada como verdadeira ou falsa.

Podemos alterar o programa visto no início do capítulo a fim de fazer com que o LED pisque apenas uma quantidade determinada de vezes e, em seguida, pare. Para isso, basta introduzir os comandos que estão dentro da função **loop( )** em um laço do tipo **while**. Para que o laço seja interrompido, o teste da condição deve resultar num valor falso após uma determinada quantidade de repetições. Assim, o programa fica como segue abaixo:

```
//Quarto programa

int contador=0;

void setup()
{
  pinMode(13, OUTPUT);
}

void loop()
{
  while(contador < 10)
  {
      digitalWrite(13, HIGH);
      delay(1000);
      digitalWite(13, LOW);
      delay(1000);
      ++contador;
  }
}
```

Nesse programa, inicialmente é atribuído o valor zero à variável contador. Dentro do laço while da função loop, a variável é testada para verificar se o seu valor é menor que 10, o que retorna verdadeiro, já que a variável contador possui zero como valor inicial. Assim, o bloco de comandos dentro das chaves é executado, fazendo com que o LED acenda, espere 1000 ms, apague, espere mais 1000 ms e, em seguida, temos o comando `++contador,` que é uma operação de incremento, fazendo com que a variável contador seja modificada para o valor atual do contador mais uma unidade, ou seja, o contador agora passa a ter o valor de 1, já que antes possuía o valor zero.

Após essa instrução, a condição do laço **while** é mais uma vez testada. Nesse caso, o contador possui o valor 1, e ainda é menor que 10, fazendo com que o teste da condição ainda retorne verdadeiro (contador<10). Então, mais uma vez o bloco dentro das chaves é executado, fazendo o LED acender e apagar mais uma vez. No final dessa segunda iteração, a variável contador é novamente incrementada, passando a ter o valor 2.

O teste da condição do laço **while** é feito mais uma vez, resultando em verdadeiro, já que 2 < 10, e o processo se repete até que a variável contador assuma o valor 10. Quando a variável contador assume o valor 10, o teste da condição **while** não é mais verdadeiro, ou seja, 10 não é menor que 10. Com isso, o bloco de comandos dentro do laço **while** não é mais executado, e o programa finaliza o laço **while**.

Como não há nenhum outro comando dentro da função loop( ), a função é encerrada e em seguida é chamada novamente. Ao retornar à função, a condição do **while** é testada mais uma vez, resultando em falso e o processo se repete indefinidamente, retornando sempre falso. Nesse caso, o LED pisca apenas 10 vezes e permanecerá apagado.

O laço do-while( )

O laço **do-while** é uma extensão do laço **while**, possuindo a seguinte forma geral:

```
do
{
   comando1;
   comando2;
   ...
}while(condição);
```

A diferença do laço **do-while** para o laço **while** é que o bloco de código compreendido entre as chaves é executado pelo menos uma vez antes da condição ser avaliada.

O laço for( )

O laço **for** é uma das mais comuns estruturas de repetição. O formato geral do laço **for** é:

```
for(inicialização; condição; atualização)
{
   Comando1;
   Comando2;
   ...
}
```

Cada uma das três seções do laço **for** possui uma função distinta, conforme explicado a seguir:

1. Inicialização: esta seção conterá uma expressão válida, utilizada normalmente para inicialização das variáveis de controle do laço **for**;
2. Condição: esta seção pode conter a condição a ser avaliada para decidir pela continuidade ou não do laço de repetição, semelhante ao laço **while**;
3. Atualização: esta seção pode conter uma ou mais declarações para atualização das variáveis de controle do laço.

O funcionamento básico do laço se dá da seguinte forma: Inicialmente, a seção de inicialização do comando é executada. Em seguida, a condição de teste é avaliada e caso seja verdadeira é executado o comando ou bloco de comandos compreendido entre as chaves. Em seguida, a seção de atualização é executada e o laço é repetido, voltando a avaliar a condição, não havendo mais inicialização. Ou seja, a inicialização ocorre apenas uma vez, no início do laço **for**.

Pode-se optar por usar apenas laços **for**, ou apenas laços **while** ou **do-while** em uma implementação. É critério do programador escolher qual a opção mais simples, ou a mais clara, no que diz respeito ao entendimento do programa.

6.8 O exemplo do LED andante

Considere que temos 8 LEDS conectados nos pinos de 0 a 7 do Arduino. Ao acender o primeiro LED e fazendo com que o LED aceso se desloque do primeiro ao último, ou seja, se o primeiro está aceso, após um tempo apaga-se esse LED enquanto acende-se o segundo, e assim sucessivamente até o oitavo LED, teremos um efeito conhecido como LED andante. Em outras palavras, estaremos fazendo o deslocamento do LED aceso.

A seguir, temos uma implementação para este problema.

```
//LED andante – primeira implementação

void setup()
{
  pinMode(0, OUTPUT);
  pinMode(1, OUTPUT);
  pinMode(2, OUTPUT);
  pinMode(3, OUTPUT);
  pinMode(4, OUTPUT);
  pinMode(5, OUTPUT);
  pinMode(6, OUTPUT);
  pinMode(7, OUTPUT);
}

void loop()
{
  int contador=0;

  while(contador <= 7)
  {
      digitalWrite(contador, HIGH);
      delay(200);
      digitalWrite(contador, LOW);
      ++contador;
  }
}
```

No exemplo acima, a função setup( ) configura os 8 pinos como saídas digitais.

Na função loop( ), inicialmente é declarada uma variável do tipo inteira chamada contador, a qual atribuímos o valor 0. No laço **while**, os comandos entre as chaves serão repetidos enquanto o contador for menor ou igual a 7. Como inicialmente o contador é igual a zero, o teste do laço **while** resulta em verdadeiro, e o bloco de comandos é executado pela primeira vez. O bloco de comandos do laço **while** executa as seguintes ações:

* O comando `digitalWrite(contador, HIGH)` leva para o nível lógico alto o pino "endereçado" pela variável contador. Nos exemplos anteriores, estávamos escrevendo o número de um pino nesse comando. Nesse caso agora, estamos usando uma variável cujo conteúdo é também um número que indica qual pino deve ser afetado pelo comando. Isso é útil pois com um mesmo comando podemos afetar diferentes pinos, bastando para isso mudarmos apenas o valor da variável;
* O comando `delay(200)` produz um atraso de 200 ms enquanto o LED estiver ligado;
* O comando `digitalWrite(contador, LOW)` leva para o nível lógico baixo o mesmo pino "endereçado" pela variável contador;
* O comando `++contador` incrementa a variável contador, que passa a ter o valor 1.

O processo se repete a partir do teste da condição do laço **while**, que retornará verdadeiro até que a variável contador assuma o valor 8, quando o teste retorna falso para a avaliação (`contador <= 7`).

Dessa forma, os oito LEDs conectados nos pinos de 0 a 7 são acesos sequencialmente e, como não existe nenhum atraso (delay) entre o comando que apaga o LED e o comando que liga o próximo LED, o efeito se apresenta como descrito inicialmente. Quando o último LED é apagado, o laço **while** se encerra. Entretanto, com a nova chamada da função loop( ), o processo inteiro se repete indefinidamente.

Como melhoria do programa, ao invés de fazer a configuração dos pinos de saída, como foi feito na função setup( ) usando 8 linhas de programa, poderíamos fazê-la usando um laço **while**, ou ainda um laço **for**. O laço mostrado abaixo, se substituído pelas 8 linhas do bloco de comandos da função setup( ), produz o mesmo efeito no programa anterior.

```
for(int contador=0; contador<=7; ++contador)
{
   pinMode(contador, OUTPUT);
}
```

Perceba que na seção de inicialização do laço **for**, é possível fazer a declaração da variável, além da atribuição do valor inicial.

Uma segunda implementação do programa usando o laço **for** na função setup( ) é apresentada a seguir:

```
//LED andante – segunda implementação

void setup()
{
  for(int contador=0; contador<=7; ++contador)
  {
     pinMode(contador, OUTPUT);
  }
}

void loop()
{
  int contador=0;

  while(contador <= 7)
  {
      digitalWrite(contador, HIGH);
      delay(200);
      digitalWrite(contador, LOW);
      ++contador;
  }
}
```

O programa acima também poderia ser escrito usando apenas laços **while** ou apenas laços **for**.

Nesse programa, a variável contador na função setup( ) é diferente da variável da função loop( ). Cada uma dessas variáveis tem escopo local às suas respectivas funções.

Para finalizar esse capítulo, perceba que, em todos os programas exemplos, os comandos compreendidos entre as chaves das funções, ou dos laços, foram escritos com uma tabulação em relação à primeira coluna. Essa característica se chama “indentação” do texto do programa, que apesar de não ser obrigatório, torna o código-fonte mais claro pois ressalta ou define a estrutura do algoritmo utilizado.■