Programação para Arduino - Primeiros Passos

Conceitos iniciais de programação para Arduino

Autor: Luís Fernando Chavier

Neste tutorial vamos apresentar os conceitos básicos de programação necessários para começar a utilizar o Arduino, e também outros tipos de de sistemas embarcados semelhantes. Para conhecer o que é possível construir com esses sistemas, veja a nossa seção de projetos. Se quiser aprender mais sobre Arduino e sistemas embarcados, explore a nossa seção de tutoriais.

Se você está usando o Arduino pela primeira vez, não deixe de ver nossos tutoriais de como configurar o Arduino para Windows ou Mac.

Aqui nós vamos explicar os conceitos de programação desde o início, e você não precisa saber nada sobre Arduino ou programação de computadores para começar. Se você já tem experiência prévia com programação, este tutorial talvez não acrescente muito ao seu conhecimento.

Nós vamos aprender como funciona um programa simples, fazendo nele algumas modificações ao longo do tutorial. Se você tiver acesso a um Arduino, você pode usá-lo ao longo do tutorial para praticar os conceitos aprendidos, tornando a experiência muito mais legal. Você só precisa de um Arduino, original ou compatível, e mais nada. Então vamos lá.

Este tutorial é dividido nas seguintes partes:

Introdução

O objetivo deste tutorial é apresentar, de uma forma simples e rápida, o básico de programação para que você possa começar a utilizar o Arduino em seus projetos, sem ter que ler muitos livros ou artigos sobre programação. O tema "desenvolvimento de software" como um todo é muito abrangente, então vamos focar apenas nos conceitos que são importantes para Arduino e sistemas embarcados em geral.

Existem muitas outras coisas para se aprender na parte de software que não vamos abordar aqui. No final do artigo nós colocamos links que você pode seguir para aprender conceitos mais avançados ou conceitos de software que não são muito utilizados na programação de sistemas embarcados.

Vamos começar explicando como funciona um computador (lembre-se que o Arduino é, no fundo, um computador).

Computador

Um **computador** é, de forma simplificada, uma máquina que processa instruções. Essas instruções são processadas no "cérebro" do computador, que se chama **microprocessador**. Todo computador possui pelo menos um microprocessador. O Arduino, por exemplo, nada mais é do que um computador muito pequeno, e ele utiliza um microprocessador do modelo **ATmega**. Alguns microprocessadores, como o ATmega, também são chamados de **microcontroladores**.

Programa de Computador

Um **programa de computador**, ou **software**, é uma sequência de instruções que são enviadas para o computador. Cada tipo de microprocessador (cérebro) entende um **conjunto de instruções** diferente, ou seja, o seu próprio "idioma". Também chamamos esse idioma de **linguagem de máquina**.

As linguagens de máquina são, no fundo, as únicas linguagens que os computadores conseguem entender, só que elas são muito difíceis para os seres humanos entenderem. É por isso nós usamos uma coisa chamada **linguagem de programação**.

No caso de sistemas como o Arduino (os chamados sistemas embarcados), o software que roda no microprocessador é também chamado de **firmware**.

Linguagem de Programação

Nós seres humanos precisamos converter as nossas idéias para uma forma que os computadores consigam processar, ou seja, a linguagem de máquina. Os computadores de hoje (ainda) não conseguem entender a linguagem natural que nós usamos no dia a dia, então precisamos de um outro "idioma" especial para instruir o computador a fazer as tarefas que desejamos. Esse "idioma" é uma linguagem de programação, e na verdade existem muitas delas.

Essas linguagens de programação também são chamadas de **linguagens de programação de alto nível**. A linguagem de programação utilizada no Arduino é a linguagem **C++** (com pequenas modificações), que é uma linguagem muito tradicional e conhecida. Essa é a linguagem que utilizaremos ao longo deste tutorial.

Para converter um programa escrito em uma linguagem de alto nível para linguagem de máquina, nós utilizamos uma coisa chamada **compilador**. A ação de converter um programa para linguagem de máquina é chamada **compilar**. Para compilar um programa, normalmente se utiliza um **ambiente de desenvolvimento** (ou IDE, do inglês *Integrated Development Environment*), que é um aplicativo de computador que possui um compilador integrado, onde você pode escrever o seu programa e compilá-lo. No caso do Arduino, esse ambiente de desenvolvimento é o Arduino IDE.

O texto contendo o programa em uma linguagem de programação de alto nível também é conhecido como o **código fonte** do programa.

Algoritmo (Programa)

Um **algoritmo**, ou simplesmente **programa**, é uma forma de dizer para um computador o que ele deve fazer, de uma forma que nós humanos conseguimos entender facilmente. Os algoritmos normalmente são escritos em linguagens de programação de alto nível. Isso se aplica a praticamente qualquer computador, inclusive o Arduino, onde um algoritmo também é conhecido como **sketch**. Para simplificar, a partir de agora nós vamos nos referir aos algoritmos, programas ou sketches simplesmente como "programas".

Um programa é composto de uma sequência de comandos, normalmente escritos em um arquivo de texto. Para este tutorial, vamos usar como base os comandos do programa mais simples do Arduino, o **Blink**, que simplesmente acende e apaga um LED, e vamos destrinchá-lo ao longo do tutorial. Veja abaixo o código fonte do Blink:

```
int led = 13;



void setup() {

  pinMode(led, OUTPUT);

}
```

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

Variável

Uma **variável** é um recurso utizado para armazenar dados em um programa de computador. Todo computador possui algum tipo de **memória**, e uma variável representa uma região da memória usada para armazenar uma determinada informação. Essa informação pode ser, por exemplo, um número, um caractere ou uma sequência de texto. Para podermos usar uma variável em um programa Arduino, nós precisamos fazer uma **declaração de variável**, como por exemplo:

```
int led;
```

Nesse caso estamos declarando uma variável do tipo `int` chamada `led`. Em seguida nós falaremos mais sobre o tipo de dado de uma variável.

Tipo de Dado

O **tipo de dado** de uma variável significa, como o próprio nome diz, o tipo de informação que se pode armazenar naquela variável. Em muitas linguagens de programação, como C++, é obrigatório definir o tipo de dado no momento da declaração da variável, como vimos na declaração da variável `led` acima. No caso dos módulos Arduino que usam processador ATmega, os tipos mais comuns de dados que utilizamos são:

- `boolean`: valor verdadeiro (`true`) ou falso (`false`)
- `char`: um caractere
- `byte`: um byte, ou sequência de 8 bits
- `int`: número inteiro de 16 bits com sinal (-32768 a 32767)
- `unsigned int`: número inteiro de 16 bits sem sinal (0 a 65535)
- `long`: número inteiro de 16 bits com sinal (-2147483648 a 2147483647)
- `unsigned long`: número inteiro de 16 bits sem sinal (0 a 4294967295)
- `float`: número real de precisão simples (ponto flutuante)
- `double`: número real de precisão dupla (ponto flutuante)
- `string`: sequência de caracteres

- `void`: tipo vazio (não tem tipo)

Para conhecer todos os tipos de dado suportados pelo Arduino, veja a seção "Data Types" nessa página.

Atribuição

Atribuir um valor a uma variável significa armazenar o valor nela para usar posteriormente. O comando de atribuição em C++ é o `=`. Para atribuírmos o valor `13` à variável `led` que criamos acima, fazemos assim:

```
led = 13;
```

Quando se armazena um valor em uma variável logo na sua inicialização, chamamos isso de **inicialização de variável**. Assim, no nosso programa de exemplo temos:

```
int led = 13;
```

O objetivo dessa linha de código é dizer que o pino 13 do Arduino será utilizado para acender o LED, e armazenar essa informação para usar depois ao longo do programa.

Os valores fixos usados no programa, como o valor `13` acima, são chamados de **constantes**, pois, diferentemente das variáveis, o seu valor não muda.

Operador

Um **operador** é um conjunto de um ou mais caracteres que serve para operar sobre uma ou mais variáveis ou constantes. Um exemplo muito simples de operador é o operador de adição, o `+`. Digamos que queremos somar dois números e atribuir a uma variável `x`. Para isso, fazemos o seguinte:

```
x = 2 + 3;
```

Após executar o comando acima, a variável `x` irá conter o valor `5`.

Cada linguagem de programação possui um conjunto de operadores diferente. Alguns dos operadores mais comuns na linguagem C++ são:

- Operadores aritméticos:
  - `+`: adição ("mais")
  - `-`: subtração ("menos")
  - `*`: multiplicação ("vezes")
  - `/`: divisão ("dividido por")
- Operadores lógicos:
  - `&&`: conjunção ("e")
  - `||`: disjunção ("ou")
  - `==`: igualdade ("igual a")

    - `!=`: desigualdade ("diferente de")
    - `!`: negação ("não")
    - `>`: "maior que"
    - `<`: "menor que"
    - `>=`: "maior ou igual a"
    - `<=`: "menor ou igual a"
- Operadores de atribuição:
    - `=`: atribui um valor a uma variável, como vimos acima.

Ao longo do desenvolvimento dos seus projetos, aos poucos você se familiarizará com todos esses operadores. Para uma lista completa, veja essa página da Wikipedia.

Função

Uma **função** é, em linhas gerais, uma sequência de comandos que pode ser reutilizada várias vezes ao longo de um programa. Para criar uma função e dizer o que ela faz, nós precisamos fazer uma **declaração de função**. Veja como uma função é declarada no nosso programa de exemplo:

```
void setup() {

  pinMode(led, OUTPUT);

}
```

Aqui estamos declarando uma função com o nome `setup()`. O que ela faz é executar os comandos de uma outra função `pinMode()`. A ação de executar os comandos de função previamente declarada é denominada **chamada de função**. Nós não precisamos declarar a função `pinMode()` porque ela já é declarada automaticamente no caso do Arduino.

Chamada de Função

Chamar uma função significa executar os comandos que foram definidos na sua declaração. Uma vez declarada, uma função pode ser chamada várias vezes no mesmo programa para que seus comandos sejam executados novamente. Para chamarmos a nossa função `setup()`, por exemplo, nós usaríamos o seguinte comando:

```
setup();
```

No entanto, no caso do Arduino, nós não precisamos chamar a função `setup()`, porque ela é chamada automaticamente. Quando compilamos um programa no Arduino IDE, ele chama a função `setup()` uma vez e depois chama a função `loop()` repetidamente até que o Arduino seja desligado ou reiniciado.

Valor de Retorno

A palavra chave que vem antes do nome da função na declaração define o tipo do **valor de retorno** da função. Toda vez que uma função é chamada, ela é executada e devolve ou **retorna** um determinado valor - esse é o valor de retorno, ou simplesmente **retorno** da função. O valor de retorno precisa ter um tipo, que pode ser qualquer um dos tipos de dados citados anteriormente. No caso da nossa função `setup()`, o tipo de retorno é `void`, o que significa que a função não retorna nada.

Para exemplificar, vamos criar uma função que retorna alguma coisa, por exemplo um número inteiro. Para retornar um valor, nós utilizamos o comando `return`:

```
int f() {

  return 1;

}
```

Quando chamada, a função `f()` acima retorna sempre o valor `1`. Você pode usar o valor de retorno de uma função para atribuí-lo a uma variável. Por exemplo:

```
x = f();
```

Após declarar a função `f()` e chamar o comando de atribuição acima, a variável `x` irá conter o valor `1`.

Parâmetros

Um outro recurso importante de uma função são os **parâmetros**. Eles servem para enviar algum dado para a função quando ela é chamada. Vamos criar por exemplo uma função que soma dois números:

```
int soma(int a, int b) {

  return a + b;

}
```

Aqui acabamos definir uma função chamada `soma()`, que aceita dois números inteiros como parâmetros. Nós precisamos dar um nome para esses parâmetros, e nesse caso escolhemos `a` e `b`. Esses parâmetros funcionam como variável que você pode usar dentro da função. Sempre que chamarmos a função `soma()`, precisamos fornecer esses dois números. O comando `return a + b;` simplesmente retorna a função com a soma dos dois números. Vamos então somar 2 + 3 e atribuir o resultado para uma variável `x`:

```
x = soma(2, 3);
```

Após a chamada acima, a variável `x` irá conter o valor `5`.

Comentários

Um **comentário** é um trecho de texto no seu programa que serve apenas para explicar (documentar) o código, sem executar nenhum tipo de comando no programa. Muitas vezes, os comentários são usados também para desabilitar comandos no código. Nesse caso, dizemos que o código foi **comentado**.

Na linguagem C++, um comentário pode ser escrito de duas formas:

- Comentário de linha: inicia-se com os caracteres `//`, tornando todo o resto da linha atual um comentário.
- Comentário de bloco: inicia-se com os caracteres `/*` e termina com os caracteres `*/`. Todo o texto entre o início e o término se torna um comentário, podendo ser composto de várias linhas.

Para facilitar a visualização, os ambientes de desenvolvimento geralmente mostram os comentários em uma cor diferente. No caso do Arduino IDE, por exemplo, os comentários são exibidos na cor cinza. Vamos então explicar o que o programa de exemplo faz, inserindo nele vários comentários explicativos:

```
/*

Programação para Arduino - Primeiros Passos

Programa de exemplo: Blink

*/



/*

Declaração da variável "led"

Indica que o LED está conectado no pino digital 13 do Arduino (D13).

*/

int led = 13;



/*

Declaração da função setup()
```

```
Esta função é chamada apenas uma vez, quando o Arduino é ligado ou rei
niciado.

*/

void setup() {

  // Chama a função pinMode() que configura um pino como entrada ou sa
ída

  pinMode(led, OUTPUT); // Configura o pino do LED como saída

}



/*

Declaração da função loop()

Após a função setup() ser chamada, a função loop() é chamada repetidam
ente até

o Arduino ser desligado.

*/

void loop() {

  // Todas as linhas a seguir são chamadas de função com passagem de p
arâmetros

  // As funções são executadas em sequência para fazer o LED acender e
apagar

  digitalWrite(led, HIGH); // Atribui nível lógico alto ao pino do LED
, acendendo-o

  delay(1000);             // Espera 1000 milissegundos (um segundo)

  digitalWrite(led, LOW);  // Atribui nível lógico baixo ao pino do LE
D, apagando-o

  delay(1000);             // Espera 1000 milissegundos (um segundo)
```

```
  // Após terminar a função loop(), ela é executada novamente repetida
s vezes,

  // e assim o LED continua piscando.

}
```

Estruturas de Controle

Estruturas de controle são blocos de instruções que alteram o fluxo de execução do código de um programa. Com elas é possível fazer coisas como executar comandos diferentes de acordo com uma condição ou repetir uma série de comandos várias vezes, por exemplo.

A seguir nós veremos algumas das estruturas de controle mais comuns usadas nas linguagens de programação em geral. Vamos também modificar o nosso programa de teste para exemplificar melhor como essas estruturas funcionam.

While

O **while** é uma estrutura que executa um conjunto de comandos repetidas vezes enquanto uma determinada condição for verdadeira. *While* em inglês quer dizer "enquanto", e pronuncia-se "uái-ou". Ele segue o seguinte formato:

```
while(condição) {

    ...

}
```

Vamos então fazer uma modificação no nosso programa para exemplificar melhor como o `while` funciona. O nosso objetivo agora é fazer o LED piscar três vezes, depois esperar cinco segundos, piscar mais três vezes e assim por diante. Nós vamos mudar o conteúdo da função `loop()` para o seguinte:

```
  // Variável para contar o número de vezes que o LED piscou

  int i = 0;



  // Pisca o LED três vezes

  while(i < 3) {

    digitalWrite(led, HIGH); // Atribui nível lógico alto ao pino do L
ED, acendendo-o
```

```
    delay(1000);              // Espera 1000 milissegundos (um segundo)
    digitalWrite(led, LOW);   // Atribui nível lógico baixo ao pino do LED, apagando-o
    delay(1000);              // Espera 1000 milissegundos (um segundo)
    i = i + 1;                // Aumenta o número de vezes que o LED piscou
  }


  delay(5000);                // Espera 5 segundos para piscar o LED de novo
```

Primeiro nós declaramos uma variável `i`. Essa variável vai contar quantas vezes o LED já piscou desde o início do programa ou desde a última pausa de cinco segundos. Nós vamos inicializar essa variável com zero porque no início da função `loop()` o LED ainda não piscou nenhuma vez sob essas condições.

Em seguida nós inserimos o comando `while`, que deve ser seguido de uma **condição** definida entre parênteses. Enquanto essa condição for verdadeira, todo o bloco de comandos entre os caracteres `{` e `}` é executado repetidamente. No caso do nosso programa, enquanto o número de "piscadas" do LED (representado pela variável `i`) for menor do que três, nós continuamos a executar os comandos que fazem o LED piscar. Isso é representado pela expressão `i < 3` dentro dos parênteses.

Entre os caracteres `{` e `}` nós colocamos o código que faz o LED piscar, como anteriormente, mas não podemos nos esquecer de somar `1` à variável que conta o número de "piscadas". Isso é feito na seguinte linha de código:

```
    i = i + 1;                // Aumenta o número de vezes que o LED piscou
```

Veja que após executar todos os comandos entre `{` e `}`, sempre teremos na variável `i` o número de vezes que o LED piscou desde o início da função `loop()`. Vamos percorrer a sequência de passos executada cada vez que a função `loop()` é chamada:

1. Atribuímos `0` à variável `i`: o LED ainda não piscou nenhuma vez.
2. Comparamos se `i < 3`: como `0` é menor do que `3`, executamos os comandos entre `{` e `}`:
    1. Executamos os comandos para acender e apagar o LED.

    2. Somamos `1` à variável `i`, tornando-a `1`: sabemos que o LED piscou uma vez.
3. Voltamos ao início do `while` e comparamos se `i < 3`: como `1` é menor do que `3`, executamos os comandos entre `{` e `}` novamente:
    1. Executamos os comandos para acender e apagar o LED.
    2. Somamos `1` à variável `i`, tornando-a `2`: sabemos que o LED piscou duas vezes.
4. Voltamos ao início do `while` e comparamos se `i < 3`: como `2` é menor do que `3`, executamos os comandos entre `{` e `}` novamente:
    1. Executamos os comandos para acender e apagar o LED.
    2. Somamos `1` à variável `i`, tornando-a `3`: sabemos que o LED piscou três vezes.
5. Voltamos ao início do `while` e comparamos se `i < 3`: como `3` **não** é menor do que `3`, não executamos mais os comandos entre `{` e `}` e prosseguimos à próxima instrução.
6. Esperamos cinco segundos por meio da chamada `delay(5000)`.

Após esses passos, chegamos ao final da função `loop()`, e como já sabemos, ela é chamada novamente pelo sistema do Arduino. Isso reinicia o ciclo, executando os passos acima indefinidamente.

Rode o programa modificado com as instruções acima no seu Arduino e tente variar o número de "piscadas" e o número

For

Agora que nós já aprendemos o comando while, fica muito fácil aprender o comando **for**, pois ele é quase a mesma coisa. Vamos modificar o conteúdo da função `loop()` como fizemos acima, porém usando o `for` no lugar do `while`:

```
  // Variável para contar o número de vezes que o LED piscou

  int i;



  // Pisca o LED três vezes

  for(i = 0; i < 3; i++) {

    digitalWrite(led, HIGH); // Atribui nível lógico alto ao pino do L
ED, acendendo-o

    delay(1000);             // Espera 1000 milissegundos (um segundo)

    digitalWrite(led, LOW);  // Atribui nível lógico baixo ao pino do
LED, apagando-o
```

```
    delay(1000);              // Espera 1000 milissegundos (um segundo)

  }



  delay(5000);                // Espera 5 segundos para piscar o LED de
novo
```

A primeira modificação que fizemos foi declarar a variável `i` sem inicializá-la com o valor `0`. Nós podemos fazer isso porque o comando `for` fará isso para a gente. Ele segue o seguinte formato:

```
for(inicialização; condição; finalização) {

    ...

}
```

Vamos descrever cada item separadamente:

- **Condição**: é uma expressão verificada repetidamente, de forma idêntica à condição entre parênteses do `while`. Enquanto ela for verdadeira, os comandos entre `{` e `}` continuam sendo executados.
- **Inicialização**: é um comando executado **apenas uma vez** no início do comando `for`.
- **Finalização**: é um comando executado **repetidas vezes** ao final de cada execução dos comandos entre `{` e `}`.

Podemos então verificar que o `for` nada mais é do que um `while` acrescido de um comando de inicialização e um comando de finalização. Para o nosso programa de teste, esses comandos são, respectivamente:

- `i = 0`: inicializa a contagem do número de "piscadas".
- `i++`: soma `1` à variável `i` ao final da execução dos comandos entre `{` e `}`; nesse caso ele é equivalente ao comando `i = i + 1`. O operador `++` é chamado de operador de **incremento**, e é muito usado na linguagem C++.

Se executarmos o programa acima no Arduino, veremos que o resultado é o mesmo que obtivemos com o programa que fizamos anteriormente utilizando o `while`.

If

O **if** é uma das estruturas mais básicas de programação em geral. *If* significa "se" em inglês, e é exatamente isso que ele faz: ele verifica uma expressão e, apenas **se** ela for **verdadeira**, executa um conjunto de comandos. Em linguagem natural, ele executa uma lógica do tipo: "**se** isso for verdadeiro, então faça aquilo"

Para ilustrar, vamos modificar o nosso programa de exemplo para que ele faça a mesma coisa que fizemos com o while e o for acima, porém vamos fazer isso usando um `if`, que segue o seguinte formato:

```
if(condição) {

    ...

}
```

A lógica é muito simples: sempre que a condição vor verdadeira, os comandos entre `{` e `}` são executados, caso contrário o programa prossegue sem executá-los. Vamos ver então como fica a função `loop()`:

```
// Variável para contar o número de vezes que o LED piscou

int i = 0;



void loop() {

  digitalWrite(led, HIGH); // Atribui nível lógico alto ao pino do LED
, acendendo-o

  delay(1000);             // Espera 1000 milissegundos (um segundo)

  digitalWrite(led, LOW);  // Atribui nível lógico baixo ao pino do LE
D, apagando-o

  delay(1000);             // Espera 1000 milissegundos (um segundo)



  i++;                     // Incrementa o número de "piscadas"

  if(i == 3) {

    delay(5000);           // Espera 5 segundos para piscar o LED de n
ovo

    i = 0;                 // Reinicia o contador de número de "piscad
as"

  }
```

```
}
```

Aqui a lógica é um pouco diferente: nós vamos manter a função `loop()` piscando o LED como no programa original, porém vamos inserir uma espera adicional de 5 segundos após cada 3 piscadas. Para isso, criamos uma variável `i` fora da função `loop()`; ela precisa ser declarada de fora da função para poder reter o seu valor entre cada execução da função `loop()`. Chamamos isso de **variável global**. Quando a variável é declarada dentro do corpo da função, ela não retém o valor etnre cada execução, sendo reiniciada a cada vez que a função é re-executada. Chamamos isso de **variável local**.

Nós usaremos então essa variável global `i` para contar, novamente, o número de vezes que o LED acendeu e apagou. Na declaração da variável, nós a inicializamos com o valor `0` para indicar que o LED não acendeu nenhuma vez ainda. A função `loop()` então começa a ser executada, acendendo e apagando o LED. Para contar o número de vezes que o LED piscou, nós adicionamos a seguinte linha de código:

```
  i++;                      // Incrementa o número de "piscadas"
```

Em seguida utilizamos o `if` para verificar se acabamos de acender o LED pela terceira vez. Para isso, usamos a expressão `i == 3` na condição do `if`. Se essa expressão for verdadeira, isso que dizer que o LED já acendeu 3 vezes, então inserimos uma pausa adicional de 5 segundos com a chamada `delay(5000)` e reiniciamos a contagem do número de "piscadas" novamente com o seguinte comando:

```
    i = 0;                  // Reinicia o contador de número de "piscad
as"
```

A partir daí a função `loop()` continua sendo chamada e o ciclo se inicia novamente.

If-Else

O **if-else**, também conhecido como **if-then-else**, pode ser visto como uma extensão do comando `if`. *Else* em inglês significa "caso contrário", e ele faz exatamente o que o nome diz: "**se** isso for verdadeiro, então faça aquilo, **caso contrário**, faça outra coisa". Ele segue o seguinte formato:

```
if(condição) {

   ...

} else {

   ...

}
```

Para exemplificar, vamos usar o programa do for que mostramos acima, mas vamos dessa vez fazer o LED acender e apagar quatro vezes antes de dar uma pausa de cinco segundos. Depois vamos fazer com que na terceira de cada uma dessas quatro "piscadas", o LED acenda por um período mais curto. Dentro da função `loop()`, teremos o seguinte:

```
  // Variável para contar o número de vezes que o LED piscou
  int i;


  // Pisca o LED três vezes
  for(i = 0; i < 3; i++) {
    if(i == 2) {
      digitalWrite(led, HIGH); // Atribui nível lógico alto ao pino do LED, acendendo-o
      delay(200);              // Espera 200 milissegundos (um segundo)
      digitalWrite(led, LOW);  // Atribui nível lógico baixo ao pino do LED, apagando-o
      delay(1800);             // Espera 1800 milissegundos (um segundo)
    } else {
      digitalWrite(led, HIGH); // Atribui nível lógico alto ao pino do LED, acendendo-o
      delay(1000);             // Espera 1000 milissegundos (um segundo)
      digitalWrite(led, LOW);  // Atribui nível lógico baixo ao pino do LED, apagando-o
      delay(1000);             // Espera 1000 milissegundos (um segundo)
    }
  }
```

```
  delay(5000);                // Espera 5 segundos para piscar o LED de
novo
```

Aqui o que fazemos é, toda vez que vamos acender o LED, verificar se é a terceira vez que isso acontece, por meio do comando `if` com a condição `i == 2`. Se essa expressão for verdadeira, isso quer dizer que já acendemos o LED duas vezes e estamos prestes a acendê-lo pela terceira vez; nesse caso mudamos o tempo que o LED fica acaso para um valor menor, de 0,2 segundo (uma redução de 0,8 segundo) e o tempo que ele fica apagado para um valor maior, de 1,8 segundos (aumento de 0,8 segundo).

Mas e se essa não for a terceira vez que o LED está sendo acionado? É aí que entra o `else`: se a condição do `if` for verdadeira, o bloco de comandos entre `{` e `}` logo após o `if` é executado, **caso contrário**, o bloco entre `{` e `}` após o `else` é executado. Isso quer dizer que para a primeira, segunda e quarta "piscadas" será usado o tempo padrão de um segundo.

Bibliotecas

As coisas que aprendemos nas seções anteriores são importantes para implementar a lógica do seu programa no Arduino, mas normalmente você vai querer fazer mais coisas além de apenas acender um LED. Quando se faz tarefas mais complexas ou se utiliza algum outro circuito conectado ao seu Arduino, um recurso muito importante são as **bibliotecas**.

Uma biblioteca é basicamente composta de código fonte adicional que você adiciona ao seu projeto por meio do comando **include**. Vejamos como adicionar, por exemplo, uma biblioteca para controle de um display de cristal liquido (LCD):

```
#include <LiquidCrystal.h>
```

Uma biblioteca do Arduino se apresenta normalmente como uma ou mais **classes** que possuem funções, os **métodos**, para acionar dispositivos, configurá-los ou executar alguma outra tarefa. Continuando com o exemplo do display de cristal líquido, para usá-lo no seu programa, primeiro é preciso inicializá-lo. O que fazemos nesse caso é criar um **objeto** para acessar o LCD (tecnicamente isso se chama **instanciar** um objeto). Vejamos como isso é feito:

```
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);
```

Quando fazemos isso, `lcd` se torna um objeto da classe `LiquidCrystal`. Isso é o equivalente a criar uma variáveldo tipo `LiquidCrystal`. Os parâmetros que são passados entre parênteses servem para inicializar a configuração desse objeto, e nesse caso correspondem aos números dos pinos que foram utilizados para conectar o LCD ao Arduino.

Quase sempre as bibliotecas de Arduino possuem um método `begin()`, que serve para fazer a configuração inicial do dispositivo que está sendo controlado. Para chamar a função `begin()` do objeto `lcd` que criamos, fazemos o seguinte:

```
lcd.begin(16, 2);
```

Normalmente esse método `begin()` é chamado de dentro da função `setup()`, ou seja, durante a inicialização do programa. Os parâmetros do método `begin()`, nesse caso, correspondem ao número de colunas e o número de linhas do LCD, respectivamente.

Feitos esses passos, já podemos escrever texto no LCD. Fazemos isso usando o método `print()` do objeto `lcd`, sempre que precisarmos ao longo do programa:

```
lcd.print("Oi!");
```

O método `print()` é apenas um dos vários métodos disponíveis na blblioteca `LiquidCrystal`. Para saber todos os métodos fornecidos por uma determinada biblioteca, é preciso consultar a documentação fornecida com ela.

Este é o processo básico de utilização de bibliotecas no Arduino. Para mais informações, leia a documentação fornecida com a biblioteca que você está usando.

Classes, objetos e métodos são conceitos de **programação orientada a objetos**. Não vamos explicar tudo em detalhes aqui, mas se você quiser aprender mais sobre isso, siga os links relacionados no final da página.

Conclusão

Neste tutorial nós vimos os conceitos básicos de programação necessários para programar um Arduino ou mesmo outras plataformas de hardware embarcado. Mas isso é só o começo, e ainda há muitas coisas a se aprender, tanto na parte de hardware quanto na parte de software. Segue então uma lista para outros artigos e tutoriais interessantes para que você possa aprender conceitos mais avançados e expandir as suas possibilidades:

- Programação em C
  - Instalando o ambiente de desenvolvimento (PUC-RS)
  - Introdução à programação C (PUC-RS)
  - Curso de Programação em C (Unicamp)
  - C Programming (em inglês)
- Programação orientada a objetos em C++
  - C++ como uma linguagem de programação orientada a objetos (Unicamp)
  - Entendendo C++ - versão original em inglês aqui
- Escrevendo bibliotecas para o Arduino (em inglês)