

Sumário

- **1. Compilação Manual (O "Por Trás dos Panos")**
 - O que o Compilador Faz?
 - Compilando com `g++`
 - Flags Essenciais
- **2. Tópicos Extras de C++ Básico**
 - Argumentos da Main (`argc`, `argv`)
 - Namespaces
 - Enums (Enumerações)
 - Unions
 - Funções Lambda
- **3. Conversão de Tipos (Casting)**
 - Conversão Implícita
 - Conversão Explícita (Manual)
 - Exemplos Práticos
- **4. `delay()` vs. `millis()`**
 - Trava: `delay()`
 - Mantem: `millis()`
- **5. Diagramas e Referências**
 - Compiladores vs. Interpretadores
 - Tipos Primitivos
 - Pinos do Arduino (PWM/ADC)

1. ☀️ Compilação Manual (O "Por Trás dos Panos")

Quando você clica em "Verificar" na Arduino IDE, uma série de processos complexos acontece. Entender isso é o que separa um programador iniciante de um engenheiro de software.

O que o Compilador Faz? (As 4 Etapas)

O processo de transformar código C++ (`.cpp`) em um executável binário (`.exe` ou `.elf`) passa por quatro fases distintas:

1. **Pré-processamento:** O compilador resolve todas as diretivas que começam com `#` (como `#include`, `#define`). Ele basicamente "copia e cola" o conteúdo dos arquivos `.h` dentro do seu arquivo e substitui os macros.
2. **Compilação:** O código C++ limpo é traduzido para **Assembly** (uma linguagem de baixo nível legível por humanos, específica para o processador).

3. **Montagem (Assembly):** O código Assembly é traduzido para **Código de Máquina** (binário puro, 0s e 1s). O resultado é um arquivo de "objeto" (geralmente terminando em `.o` ou `.obj`).
4. **Linkagem (Linking):** O **Linker** junta todos os arquivos `.o` do seu projeto e as bibliotecas necessárias em um único arquivo executável final.

Compilando com `g++`

O `g++` é o comando para invocar o compilador C++ do projeto GNU (GCC).

Compilando um arquivo único

Se o seu projeto for apenas um arquivo:

```
# Sintaxe: g++ [arquivo_fonte] -o [nome_do_executavel]
g++ main.cpp -o meu_foguete
```

- Isso gera um arquivo `meu_foguete.exe` (Windows) ou `meu_foguete` (Linux/Mac).

Compilando múltiplos arquivos

Se você separou seu código (como recomendado na aula de Classes):

```
# Liste todos os arquivos .cpp que você quer compilar juntos
g++ main.cpp SerialReader.cpp StrobeLight.cpp -o sistema_voo
```

Flags Essenciais

As "flags" são opções que passamos ao compilador para alterar seu comportamento.

- **-o nome (Output):** Define o nome do arquivo final. Se você não usar, ele gera um arquivo chamado `a.out` ou `a.exe`.
- **-c (Compile only):** Realiza apenas as etapas 1, 2 e 3 (gera o arquivo `.o`), mas **não** faz a linkagem. Útil para compilar bibliotecas ou verificar erros de sintaxe sem criar o executável.

```
g++ -c main.cpp # Gera main.o
```

- **-I /caminho (Include Path):** Diz ao compilador onde procurar por arquivos de cabeçalho (`.h`) que não estão na pasta padrão.

```
# "Procure por .h também na pasta 'include/'"
g++ main.cpp -I ./include -o app
```

- **-L /caminho (Library Path):** Diz ao **Linker** onde procurar por arquivos de biblioteca compilados (.lib ou .a).
- **-l nome (Link Library):** Diz ao **Linker** o nome específico da biblioteca para usar.
 - *Nota:* Se a biblioteca chama `libmath.a`, você usa apenas `-lmath` (o prefixo 'lib' e a extensão são omitidos).

2. Tópicos Extras de C++ Básico

Conceitos que não são exclusivos do Arduino, mas fundamentais para C++.

Argumentos da Main (`argc`, `argv`)

Em programas de PC (console), a função `main` pode receber dados quando o programa inicia.

```
int main(int argc, char* argv[]) { ... }
```

- **int argc (Argument Count):** O número de argumentos passados.
- **char* argv[] (Argument Vector):** Um array de strings com os argumentos.
- *Exemplo:* Se você rodar `./foguete -teste`, então `argc` será 2, `argv[0]` será `./foguete` e `argv[1]` será `-teste`.

Teste:

```
#include <iostream>
using namespace std;

int main(int argc, char* argv[]) {
    cout << "Número de argumentos: " << argc << "\n";
    for(int i = 0; i < argc; i++){
        cout << "Argumento " << i << ": " << argv[i] << "\n";
    }
    return 0;
}
```

Namespaces

Servem para organizar o código e evitar conflitos de nomes (colisões). Imagine que você tem duas bibliotecas diferentes e ambas possuem uma função chamada `iniciar()`.

Como criar e usar:

```
// Definindo o namespace da equipe de Motores
namespace Motor {
    void iniciar() {
```

```

        // Lógica para acender o motor
    }

}

// Definindo o namespace da equipe de Sensores
namespace Sensor {
    void iniciar() {
        // Lógica para ligar o acelerômetro
    }
}

int main() {
    // iniciar(); // ERRO! O compilador não sabe qual chamar.

    // Uso correto: Especificando o sobrenome (namespace)
    Motor::iniciar();
    Sensor::iniciar();

    return 0;
}

```

O caso do std:

- **O namespace std:** A biblioteca padrão do C++ (`cout`, `string`, `vector`) vive dentro do namespace `std`.
- **using namespace std;:** Isso "despeja" tudo do `std` no seu código global.
 - *Por que pode ser ruim?* Se você criar uma variável chamada `cout` ou uma função `max` e usar `using namespace std;`, o compilador ficará confuso entre a sua versão e a do sistema. Em projetos grandes, prefira usar `std::cout` explicitamente.

Enums (Enumerações)

Uma forma de criar constantes nomeadas que deixam o código muito mais legível. É a ferramenta perfeita para controlar **Máquinas de Estado**.

Exemplo com Máquina de Estados:

```

#include <iostream>

// O compilador numera automaticamente: 0, 1, 2, 3...
enum EstadoVoo {
    PREPARACAO,
    LANCAMENTO,
    APOGEU,
    RECUPERACAO
};

int main() {
    EstadoVoo estadoAtual = LANCAMENTO;

    // O switch torna a lógica de controle muito clara

```

```
switch (estadoAtual) {
    case PREPARACAO:
        std::cout << "Verificando sensores..." << std::endl;
        break;
    case LANCAMENTO:
        std::cout << "Fogo! O foguete subiu." << std::endl;
        break;
    case APOGEU:
        std::cout << "Altitude maxima. Abrir paraquedas." << std::endl;
        break;
    case RECUPERACAO:
        std::cout << "Descendo..." << std::endl;
        break;
}
return 0;
}
```

Unions

São parecidos com `structs` na sintaxe, mas funcionam de forma radicalmente diferente na memória. Em uma `union`, todos os membros compartilham o **mesmo endereço de memória**.

- **Struct:** O tamanho é a **soma** dos membros (Guarda A **E** B).
- **Union:** O tamanho é o do **maior** membro (Guarda A **OU** B).

Exemplo Prático (Economia de Memória): Imagine que seu rádio envia pacotes de dados, mas cada pacote carrega apenas um tipo de informação por vez: ou é um comando (inteiro) ou é a altitude (float).

```
#include <iostream>

union DadosPacote {
    int comandoID;      // 4 bytes
    float altitude;    // 4 bytes
    // Tamanho total da Union: 4 bytes (eles dividem o mesmo espaço)
};

int main() {
    DadosPacote pacote;

    // Cenário 1: Enviando um comando
    pacote.comandoID = 10;
    std::cout << "Comando: " << pacote.comandoID << std::endl;

    // Cenário 2: Enviando altitude
    pacote.altitude = 500.5;
    std::cout << "Altitude: " << pacote.altitude << std::endl;

    // CUIDADO! Se tentarmos ler o ID agora, teremos lixo,
    // pois os bytes da altitude sobrescreveram os bytes do ID.
    std::cout << "Comando (Lixo): " << pacote.comandoID << std::endl;
}
```

```
    return 0;  
}
```

Funções Lambda

São funções "anônimas" (sem nome) que você pode escrever direto dentro de outra função. São muito usadas em algoritmos modernos.

Sintaxe: [captura](parametros){ corpo }

```
#include <iostream>  
using namespace std;  
  
int main() {  
    // Exemplo simples  
    int x = 2;  
    auto soma = [x](int a, int b) {  
        // O 'x' foi capturado do lado de fora  
        return a + b + x;  
    };  
  
    int resultado = soma(5, 2); // 5 + 2 + 2 = 9  
    cout << resultado << endl;  
    return 0;  
}
```

3. Conversão de Tipos (Casting)

Em C++, muitas vezes precisamos transformar um tipo de dado em outro (por exemplo, transformar um número `float` vindo de um sensor em um `int` para economizar espaço, ou interpretar um `char` como um número). Isso se chama **Casting**.

[!WARNING] ⚠️ **OBSERVAÇÃO CRÍTICA: string NÃO é um tipo primitivo!**

Diferente de `int`, `float`, `char` ou `bool`, o tipo `string` (seja `std::string` no PC ou `String` no Arduino) é uma **Classe (Objeto)**.

Isso significa que **você NÃO pode fazer casting direto** de uma string para número.

- ✗ **Errado:** `int valor = (int)"123";` (Isso não converte o texto em número, tenta converter o endereço de memória).
- ☑ **Certo:** Você deve usar métodos da classe, como `"123".toInt()` (no Arduino) ou `std::stoi("123")` (no PC).

1. Conversão Implícita (Automática)

O compilador faz automaticamente quando não há risco de perda de dados grave, geralmente promovendo um tipo menor para um maior.

```
int a = 10;
float b = a; // 0 10 vira 10.0 automaticamente.
```

2. Conversão Explícita (Casting Manual)

Quando você quer forçar a conversão. Existem dois jeitos principais:

A. C-Style Cast (O jeito "Raiz")

É herdado da linguagem C. É rápido e muito comum no mundo Arduino, mas é considerado "força bruta" porque o compilador tenta obedecer a qualquer custo, o que pode mascarar erros.

Sintaxe: `(novo_tipo) valor`

```
float temperatura = 25.8;
int leitura = (int) temperatura;
// Resultado: 25 (A parte decimal .8 é truncada/cortada, não arredondada!)
```

B. Static Cast (O jeito C++ Moderno)

É a forma recomendada em C++ moderno. É mais seguro porque o compilador verifica se a conversão faz sentido antes de aceitar.

Sintaxe: `static_cast<novo_tipo>(valor)`

```
float temperatura = 25.8;
int leitura = static_cast<int>(temperatura);
// Resultado: 25
```

💻 Exemplos Práticos

Exemplo 1: A Pegadinha da Divisão

Se você dividir dois inteiros, o resultado será inteiro, mesmo que a variável de destino seja `float`. O casting resolve isso.

```
int a = 5;
int b = 2;

// SEM casting:
float resultadoRuim = a / b;
// O computador faz 5/2 = 2 (inteiro) e depois guarda 2.0. ERRADO.
```

```
// COM casting:  
float resultadoBom = (float)a / b;  
// Agora ele faz 5.0 / 2 = 2.5. CORRETO.
```

Exemplo 2: Char para Int (ASCII)

Todo caractere é, no fundo, um número da tabela ASCII.

```
char letra = 'A';  
int codigoAscii = (int) letra;  
  
// codigoAscii será 65 (o valor de 'A' na tabela ASCII)  
// Isso é muito útil para criptografia simples ou comunicação serial.
```

4. ⏳ `delay()` vs. `millis()`

Em programação embarcada (e especialmente em aviônica), a forma como você lida com o tempo define se o seu foguete funciona ou falha.

Trava: `delay()`

A função `delay(1000)` é a primeira que aprendemos, mas é a **pior** para sistemas reais.

- **O que ela faz:** Ela congela o processador. Ela diz: "*Pare TUDO o que estiver fazendo e fique parado olhando para o relógio por 1 segundo.*"
- **O Problema:** Enquanto o Arduino está no `delay()`, ele é **cego, surdo e mudo**. Ele não lê sensores, não verifica se o botão de emergência foi apertado e não ajusta a trajetória. Em um foguete a 500 km/h, 1 segundo "cego" é uma eternidade.

Exemplo Ruim (Bloqueante):

```
// Pisca o LED, mas TRAVA tudo  
digitalWrite(LED_BUILTIN, HIGH);  
delay(1000); // <--- O processador dorme aqui por 1s  
digitalWrite(LED_BUILTIN, LOW);  
delay(1000); // <--- Dorme de novo. Nada mais roda.
```

Mantém: `millis()`

A função `millis()` é a chave para a **multitarefa** (fazer várias coisas "ao mesmo tempo").

- **O que ela faz:** Ela apenas retorna um número: **quantos milissegundos se passaram desde que o Arduino ligou**. Chamar `millis()` é instantâneo, como olhar rapidamente para o seu relógio de pulso.

- A Lógica:** Em vez de parar e esperar (`delay`), você usa a lógica de "**Verificação de Tempo**". Você anota a hora em que fez algo pela última vez e, a cada volta do `loop`, verifica: "Já passou tempo suficiente?". Se não, você continua trabalhando em outras coisas.

Analogia da Cozinha:

- Com `delay()`:** Você coloca o bolo no forno e fica **parado na frente dele por 40 minutos**, sem fazer mais nada, até apitar.
- Com `millis()`:** Você anota a hora que colocou o bolo. Aí você vai lavar a louça, cortar legumes... A cada minuto, você **dá uma olhada rápida no relógio**. Se passou 40 min, você tira o bolo. Se não, você continua trabalhando.

Exemplo Bom (Não-Bloqueante):

```
unsigned long tempoAnterior = 0; // "Anotação" da última vez
const long intervalo = 1000;      // Quanto tempo esperar

void loop() {
    // 1. Pega a hora atual (olhada rápida no relógio)
    unsigned long tempoAtual = millis();

    // 2. Verifica: "Já passou 1 segundo desde a última vez?"
    if (tempoAtual - tempoAnterior >= intervalo) {

        // Sim! Faz a tarefa (inverte o LED)
        // ... código para piscar o LED ...

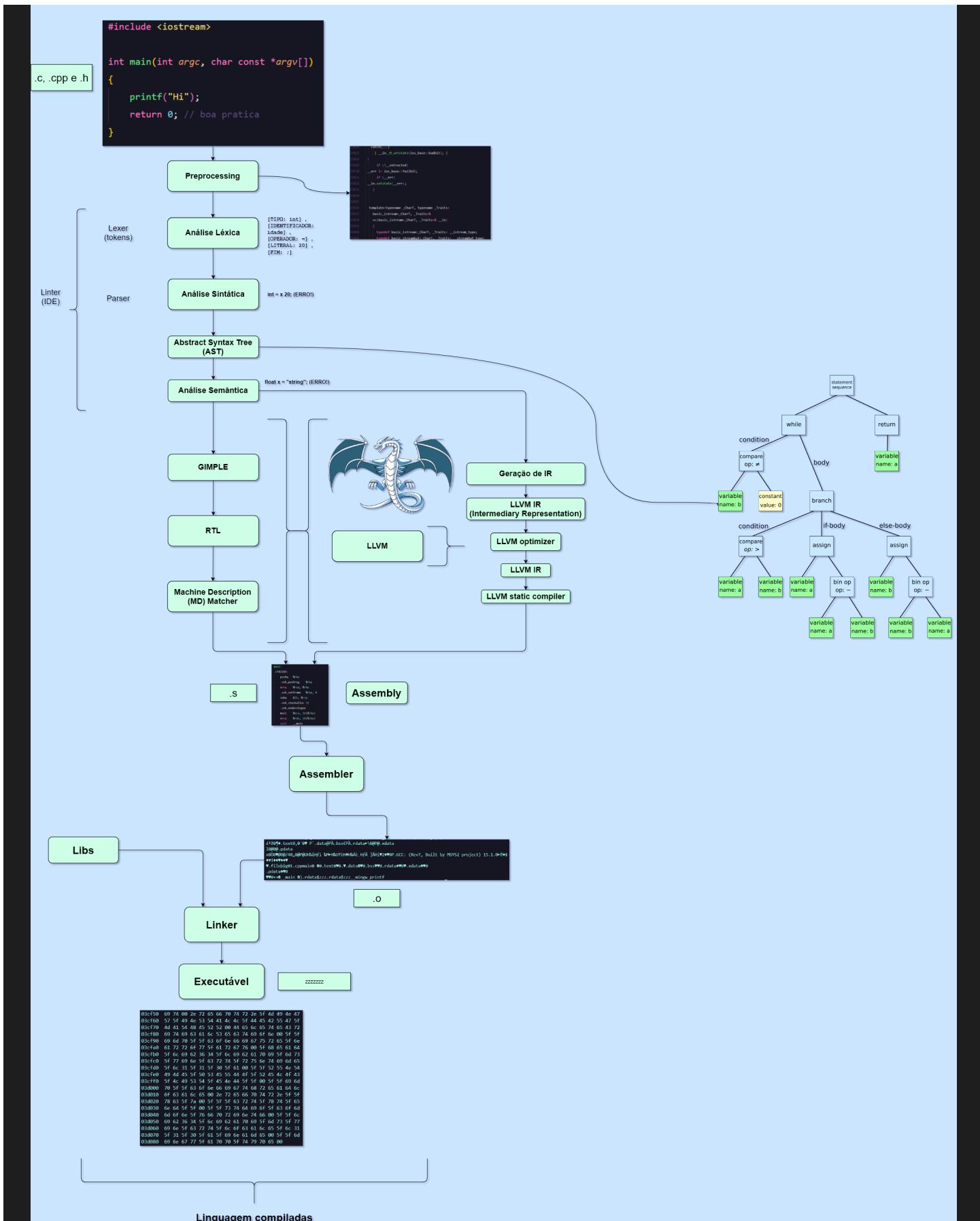
        // 3. Atualiza a anotação para agora
        tempoAnterior = tempoAtual;
    }

    // 4. O loop continua LIVRE para fazer outras coisas aqui!
    lerSensores(); // Isso roda milhares de vezes enquanto espera o LED piscar
}
```

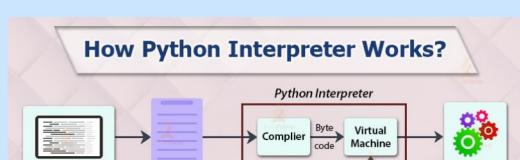
5. Diagramas e Referências

Compiladores vs. Interpretadores

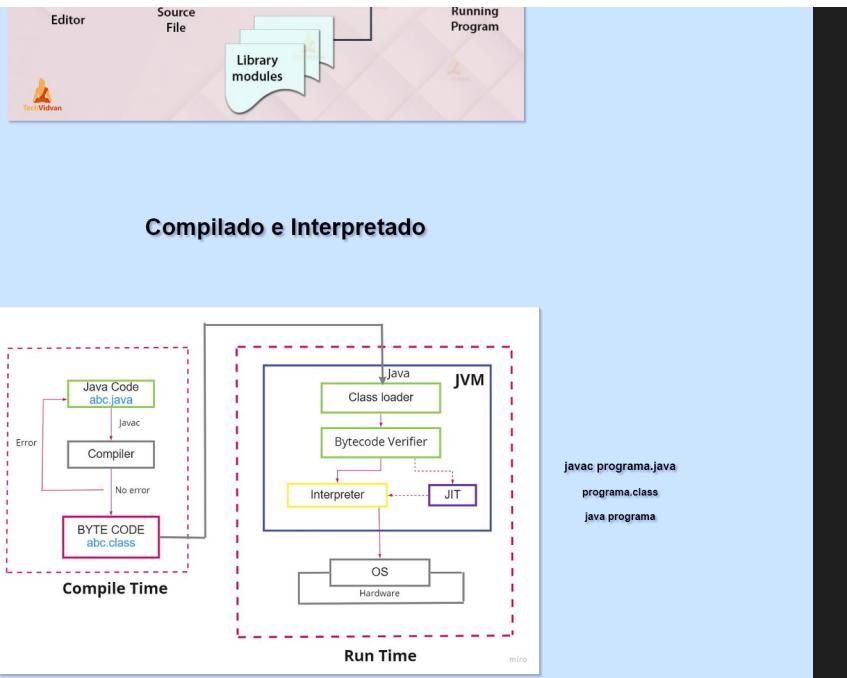




Compiladores vs. Interpretadores



```
python programa.py
```



Tipos Primitivos

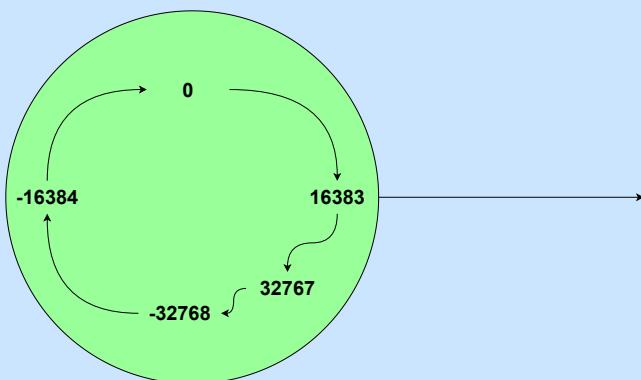
Tipos Primitivos

Sequência	Bytes ocupados	Faixa (ou intervalo)
short int signed int signed short int wchart_t signed wchar_t	2	-32.768 a 32.767
unsigned short int unsigned wchar_t	2	0 a 65.535
int long int signed long int	4	-2.147.483.648 até 2.147.483.647
unsigned int unsigned long int	4	0 a 4.294.967.295
char signed char	1	-128 a 127
unsigned char	1	0 a 255
bool	1	true ou false
float	4	1,2e-38 a 3,4e+38
double long float	8	2,2e-308 a 1,8e+308

$$1 \quad 1 \quad 0 \quad 1 \quad 1_2$$

$$2^4 + 2^3 + 0 + 2^1 + 2^0 = 16 + 8 + 2 + 1 = 27_{10}$$

Overflow



Soma de binários

Com valores negativos

Complemento de Dois

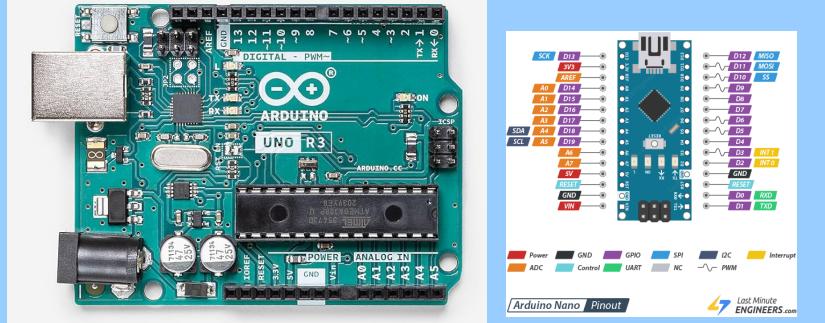
complemento de 2									
	0	0	1	0	1	0	1	0	
+	1	1	0	1	0	1	¹⁰⁰¹¹¹ 0	1	1
(-42) ₁₀ =	1	1	0	1	0	1	1	0	

$$2^{\overbrace{14}^{01111111111111}} + 2^{\overbrace{13}^0} + \dots + 2^0 = \frac{1 \cdot (2^{15} - 1)}{(2 - 1)} = 32767$$

$$\overbrace{10000000000000}^{\text{15}} - 2^{15} + 0 = -32768$$

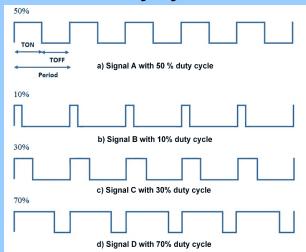
Pinos do Arduino (Funções e PWM/ADC)

Função	Pinos Digitais Comuns(ex: 2, 4, 7, 8)	Pinos PWM (com ~) (ex: -3, -5, -9)	Pinos Analógicos(ex: A0, A1, A2)
digitalRead()	<input checked="" type="checkbox"/> Sim	<input checked="" type="checkbox"/> Sim	<input checked="" type="checkbox"/> Sim
digitalWrite()	<input checked="" type="checkbox"/> Sim	<input checked="" type="checkbox"/> Sim	<input checked="" type="checkbox"/> Sim
analogRead()	<input checked="" type="checkbox"/> Não	<input checked="" type="checkbox"/> Não	<input checked="" type="checkbox"/> Sim (Função Principal)
analogWrite()	<input checked="" type="checkbox"/> Não	<input checked="" type="checkbox"/> Sim (Função Principal)	<input checked="" type="checkbox"/> Não



PWM

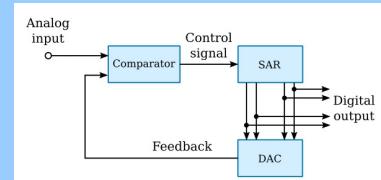
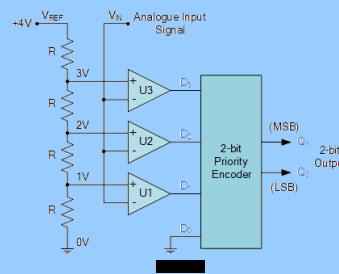
Duty Cycle



0 até 255

ADC (Analog-to-Digital Converter)

Arduíno (10 bits): 0 até 1023



Característica	PWM (Saída)	ADC (Entrada)
Direção	Digital -> "Falso Analógico" (Média)	Analógico Real -> Digital (Número)
Como funciona	Corta o tempo (Tempo ligado vs Desligado)	Corta a voltagem (Mede altura em degraus)
O Físico	Aproveita a inércia (do motor/olho) para criar média.	Usa capacitores e comparadores para medir nível.
Uso	Controlar velocidade de motor, brilho de LED.	Ler sensores de temperatura, som, luz.