



Flávio Copes

—

O Manual C

Índice

- [Índice](#)
- [1. Introdução ao C](#)
- [2. Variáveis e tipos](#)
- [2.1. Números inteiros](#)
- [2.2. Inteiros não assinados](#)
- [2.3. O problema do estouro](#)
- [2.4. Advertências ao declarar o tipo errado](#)
- [2.5. Números de ponto flutuante](#)
- [3. Constantes](#)
- [4. Operadores](#)
- [4.1. Operadores aritméticos](#)
- [4.2. Operadores de comparação](#)
- [4.3. Operadores lógicos](#)
- [4.4. Operadores de atribuição composta](#)
- [4.5. Operadores diversos](#)
- [4.5.1. O operador ternário](#)
- [4.5.2. Dimensão](#)
- [4.6. Precedência do operador](#)
- [5. Condicionais](#)
- [5.1. Se](#)
- [5.2. Interruptor](#)
- [6. Loops](#)
- [6.1. Para loops](#)
- [6.2. Enquanto loops](#)
- [6.3. Fazer enquanto loops](#)



St Adobe Stock

Obtenha 10 fotos gratuitas do Adobe Stock. Comece a baixar incríveis fotos royalty-free hoje mesmo.

ANÚNCIOS VIA CARBON

- [6.4. Quebrando um loop usando a quebra](#)
- [7. Matrizes](#)
- [8. Cordas](#)
- [9. Ponteiros](#)
- [10. Funções](#)
- [11. Entrada e Saída](#)
- [11.1. printf\(\)](#)
- [11.2. scanf\(\)](#)
- [12. Âmbito das variáveis](#)
- [13. Variáveis estáticas](#)
- [14. Variáveis globais](#)
- [15. Definições de tipo](#)
- [16. Tipos enumerados](#)
- [17. Estruturas](#)
- [18. Parâmetros de linha de comando](#)
- [19. Arquivos de cabeçalho](#)
- [20. O pré-processador](#)
- [20.1. Condicionais](#)
- [20.2. Constantes simbólicas](#)
- [20.3. Macros](#)
- [20.4. Se definido](#)
- [20.5. Constantes simbólicas predefinidas que você pode usar](#)

1. Introdução ao C

C é provavelmente a linguagem de programação mais conhecida. É usada como a linguagem de referência para cursos de ciência da computação em todo o mundo, e é provavelmente a linguagem que as pessoas mais aprendem na escola entre Python e Java.

Lembro-me de ser a minha segunda linguagem de programação de sempre, depois de Pascal.

C não é apenas o que os alunos usam para aprender programação. Não é uma linguagem acadêmica. E eu diria que não é a linguagem mais fácil, porque C é uma linguagem de programação de nível bastante baixo.

Hoje, o C é amplamente utilizado em dispositivos embarcados e alimenta a maioria dos servidores da Internet, que são construídos usando Linux. O kernel Linux é construído usando C, e isso também significa que C alimenta o núcleo de todos os dispositivos Android. Podemos dizer que o código C percorre uma boa parte do mundo inteiro. Agora. Bastante notável.

Quando foi criado, C era considerado uma linguagem de alto nível, porque era portátil entre máquinas. Hoje nós meio que damos como certo que podemos executar um programa escrito em um Mac no Windows ou Linux, talvez usando Node.js ou Python. Era uma vez, este não era o caso. O que C trouxe para a mesa foi uma linguagem simples de implementar, tendo um compilador que poderia ser facilmente portado para diferentes máquinas.

Eu disse compilador: C é uma linguagem de programação compilada, como Go, Java, Swift ou Rust. Outras linguagens de programação populares como Python, Ruby ou JavaScript são interpretadas. A diferença é consistente: uma linguagem compilada gera um arquivo binário que pode ser executado e distribuído diretamente.

C não é lixo recolhido. Isso significa que temos que gerenciar a memória nós mesmos. É uma tarefa complexa e que requer muita atenção para evitar bugs, mas também é o que torna o C ideal para escrever programas para dispositivos embarcados como o Arduino.

C não esconde a complexidade e as capacidades da máquina por baixo. Você tem muito poder, uma vez que você sabe o que você pode fazer.

Quero apresentar o primeiro programa C agora, que chamaremos de "Olá, Mundo!"

Olá.c

```
#include <stdio.h>

int main(void) {
    printf("Hello, World!");
}
```

Vamos descrever o código-fonte do programa: primeiro importamos a biblioteca (o nome significa biblioteca de entrada-saída padrão).stdio

Esta biblioteca nos dá acesso a funções de entrada/saída.

C é uma linguagem muito pequena em sua essência, e qualquer coisa que não faça parte do núcleo é fornecida por bibliotecas. Algumas dessas bibliotecas são construídas por programadores normais e disponibilizadas para outros usarem. Algumas outras bibliotecas são incorporadas ao compilador. Como e outros.stdio

stdio são as bibliotecas que fornecem a função.printf()

Esta função é encapsulada em uma função. A função é o ponto de entrada de qualquer programa C.main()main()

Mas o que é uma função, afinal?

Uma função é uma rotina que usa um ou mais argumentos e retorna um único valor.

No caso de , a função não obtém argumentos e retorna um inteiro. Identificamos isso usando a palavra-chave para o argumento e a palavra-chave para o valor de retorno.main()voidint

A função tem um corpo, que é envolto em aparelhos encaracolados, e dentro do corpo temos todo o código que a função precisa para realizar suas operações.

A função é escrita de forma diferente, como você pode ver. Ele não tem nenhum valor de retorno definido e passamos uma cadeia de caracteres, encapsulada em aspas duplas. Não especificamos o tipo de argumento.printf()

Isso porque essa é uma invocação de função. Em algum lugar, dentro da biblioteca, é definido como stdio.printf

```
int printf(const char *format, ...);
```

Você não precisa entender o que isso significa agora, mas, em suma, essa é a definição e quando chamamos , é aí que a função é executada.

```
printf("Hello, World!");
```

A função que definimos acima:`main()`

```
#include <stdio.h>

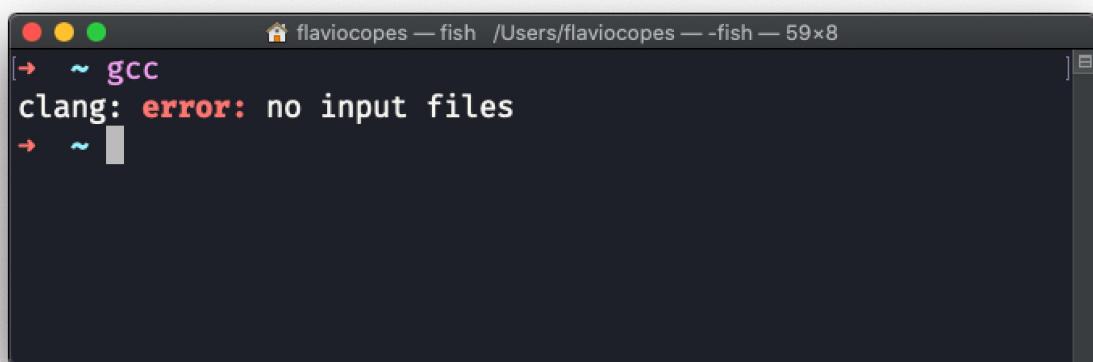
int main(void) {
    printf("Hello, World!");
}
```

será executado pelo sistema operacional quando o programa for executado.

Como executamos um programa C?

Como mencionado, C é uma linguagem compilada. Para executar o programa, devemos primeiro compilá-lo. Qualquer computador Linux ou macOS já vem com um compilador C embutido. Para Windows, você pode usar o Subsistema Windows para Linux (WSL).

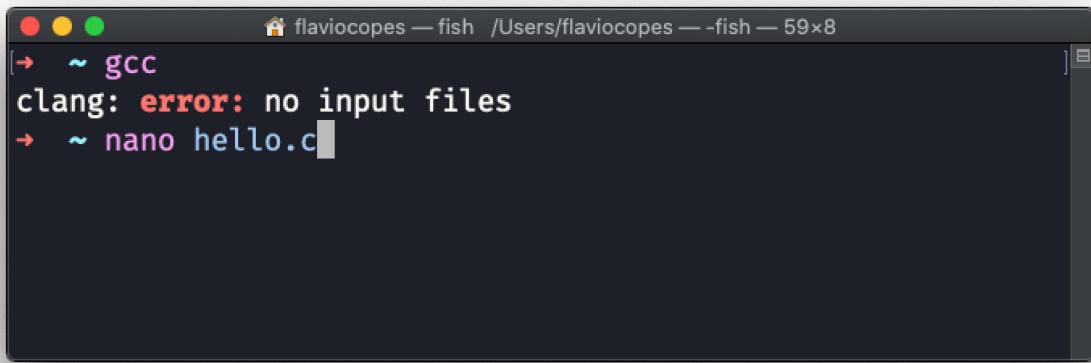
Em qualquer caso, quando você abre a janela do terminal, você pode digitar , e este comando deve retornar um erro dizendo que você não especificou nenhum arquivo:`gcc`



The screenshot shows a dark-themed terminal window. The title bar reads "flaviocopes — fish /Users/flaviocopes — -fish — 59x8". In the terminal, the user has typed "gcc" and pressed enter. The output is: "clang: error: no input files". The cursor is visible at the end of the command line.

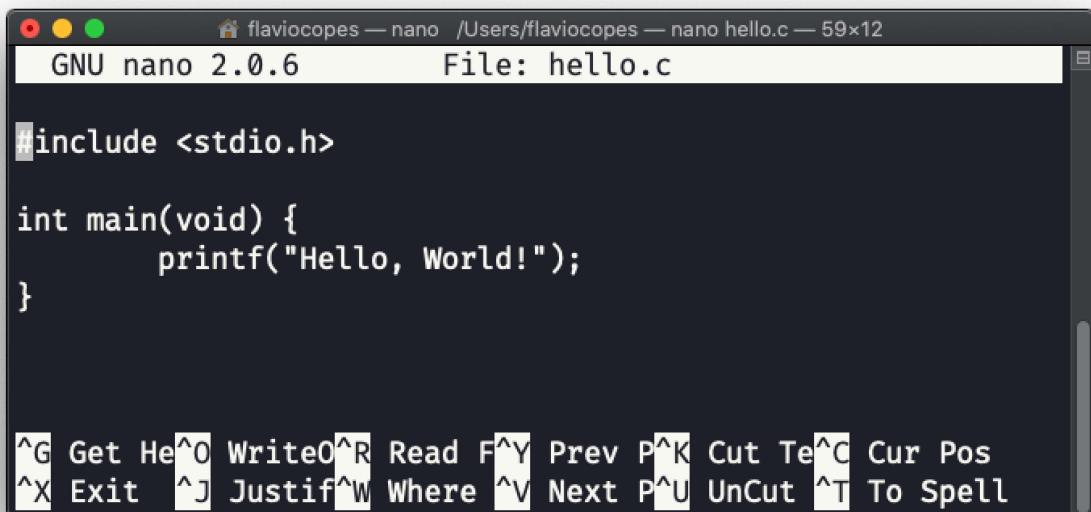
Isso é bom. Isso significa que o compilador C está lá, e podemos começar a usá-lo.

Agora digite o programa acima em um arquivo. Você pode usar qualquer editor, mas por uma questão de simplicidade eu vou usar o editor na linha de comando: hello.c nano



```
flaviocopes — fish /Users/flaviocopes — -fish — 59x8
[→ ~ gcc
clang: error: no input files
→ ~ nano hello.c]
```

Digite o programa:



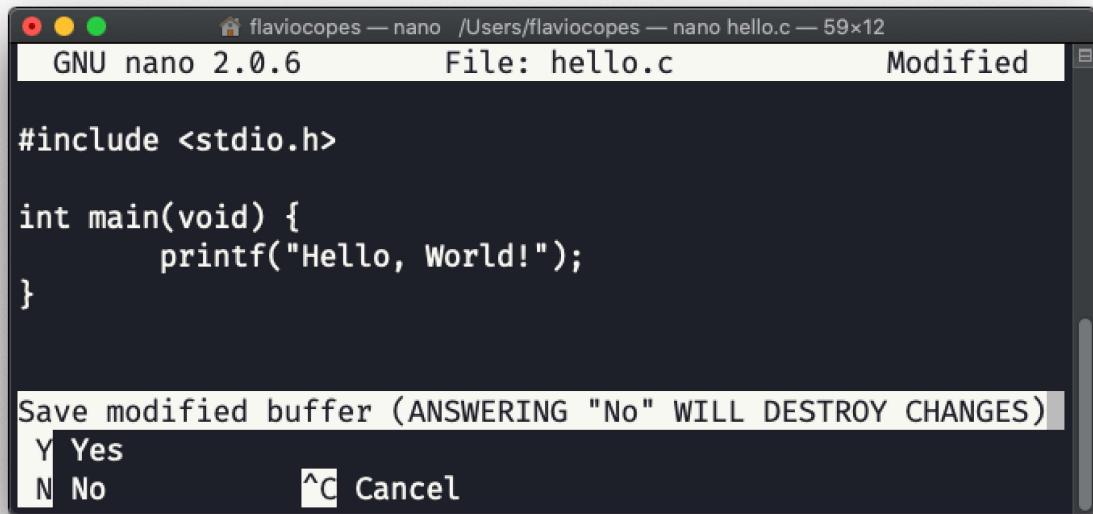
```
flaviocopes — nano /Users/flaviocopes — nano hello.c — 59x12
GNU nano 2.0.6          File: hello.c

#include <stdio.h>

int main(void) {
    printf("Hello, World!");
}

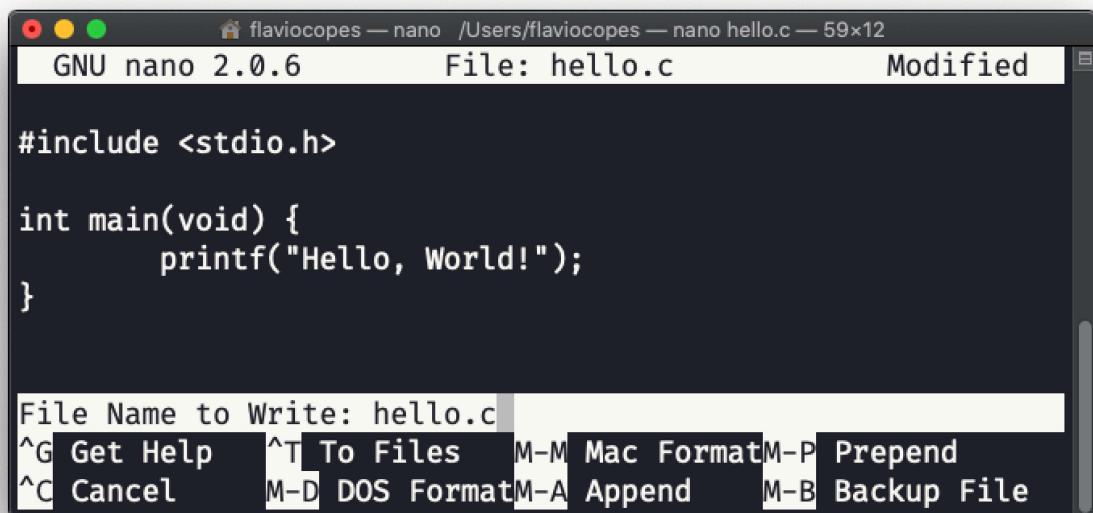
^G Get He^O Write^R Read F^Y Prev P^K Cut Te^C Cur Pos
^X Exit ^J Justif^W Where ^V Next P^U UnCut ^T To Spell
```

Agora pressione para sair:ctrl-X



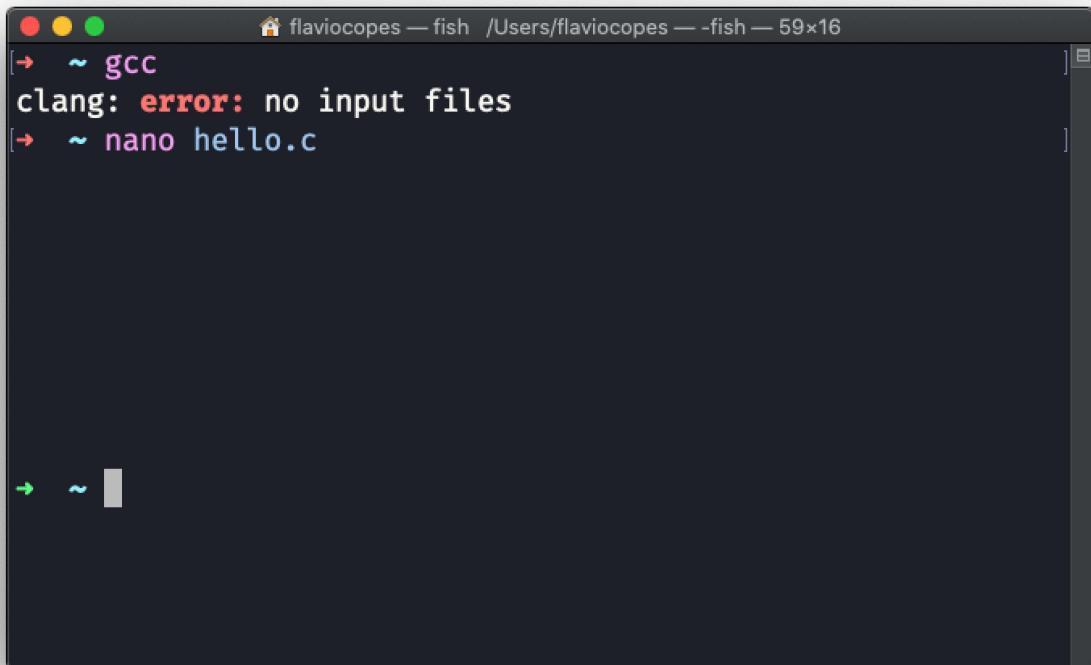
```
GNU nano 2.0.6          File: hello.c          Modified\n\n#include <stdio.h>\n\nint main(void) {\n    printf(\"Hello, World!\");\n}\n\nSave modified buffer (ANSWERING \"No\" WILL DESTROY CHANGES)\nY Yes\nN No          ^C Cancel
```

Confirme pressionando a tecla e pressione Enter para confirmar o nome do arquivo:y



```
GNU nano 2.0.6          File: hello.c          Modified\n\n#include <stdio.h>\n\nint main(void) {\n    printf(\"Hello, World!\");\n}\n\nFile Name to Write: hello.c\n^G Get Help  ^T To Files  M-M Mac Format M-P Prepend\n^C Cancel    M-D DOS Format M-A Append   M-B Backup File
```

É isso, devemos estar de volta ao terminal agora:



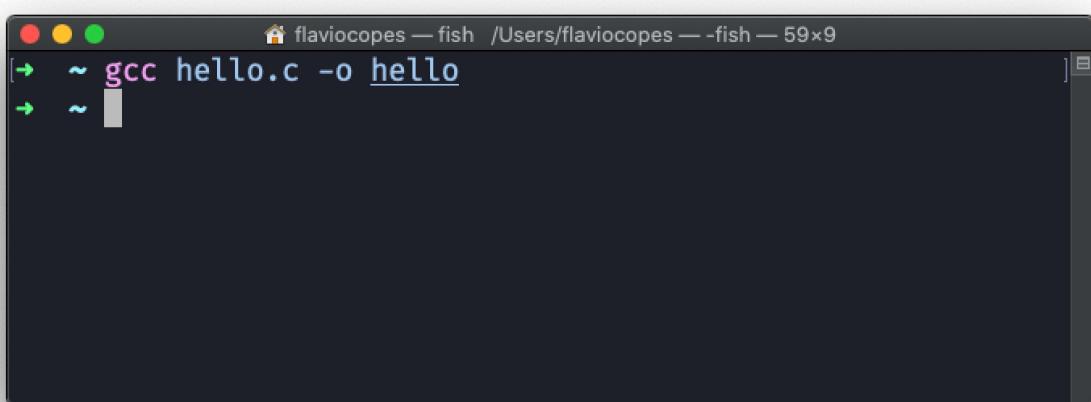
```
flaviocopes — fish /Users/flaviocopes — -fish — 59x16
[→ ~ gcc
clang: error: no input files
[→ ~ nano hello.c

→ ~ ]
```

Agora digite

```
gcc hello.c -o hello
```

O programa não deve lhe dar erros:

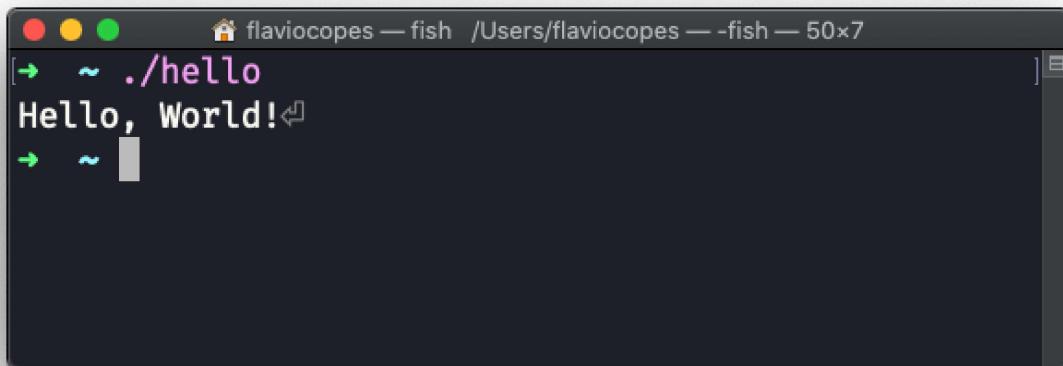


```
flaviocopes — fish /Users/flaviocopes — -fish — 59x9
[→ ~ gcc hello.c -o hello
[→ ~ ]
```

mas deveria ter gerado um executável. Agora digite `hello`

```
./hello
```

para executá-lo:

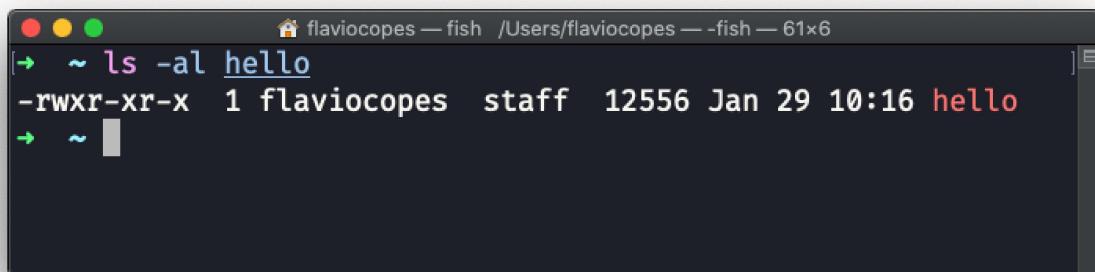


A screenshot of a macOS terminal window titled "flaviocopes — fish /Users/flaviocopes — -fish — 50x7". The terminal shows the command `./hello` being run, followed by the output "Hello, World!". The terminal has a dark theme with red, yellow, and green window controls.

Eu prepend para o nome do programa, para dizer ao terminal que o comando está na pasta atual. ./

Incrível!

Agora, se você chamar `ls`, você pode ver que o programa tem apenas 12KB de tamanho:
`ls -al hello`



A screenshot of a macOS terminal window titled "flaviocopes — fish /Users/flaviocopes — -fish — 61x6". The terminal shows the command `ls -al hello` being run, displaying the file information for "hello". The output shows the file is executable (-rwxr-xr-x), owned by flaviocopes and staff, has a size of 12556 bytes, and was modified on Jan 29 at 10:16. The terminal has a dark theme with red, yellow, and green window controls.

Este é um dos prós do C: é altamente otimizado, e esta também é uma das razões pelas quais é tão bom para dispositivos incorporados que têm uma quantidade muito limitada de recursos.

2. Variáveis e tipos

C é uma linguagem tipada estaticamente.

Isso significa que qualquer variável tem um tipo associado, e esse tipo é conhecido em tempo de compilação.

Isso é muito diferente de como você trabalha com variáveis em Python, JavaScript, PHP e outras linguagens interpretadas.

Quando você cria uma variável em C, você precisa especificar o tipo de uma variável na declaração.

Neste exemplo, inicializamos uma variável com o tipo :ageint

```
int age;
```

Um nome de variável pode conter qualquer letra maiúscula ou minúscula, pode conter dígitos e o caractere de sublinhado, mas não pode começar com um dígito. e são nomes de variáveis válidos, não é.AGEAge101age

Você também pode inicializar uma variável na declaração, especificando o valor inicial:

```
int age = 37;
```

Depois de declarar uma variável, você poderá usá-la em seu código de programa e poderá alterar seu valor a qualquer momento, usando o operador, por exemplo, como em , desde que o novo valor seja do mesmo tipo.=age = 100;

Neste caso:

```
#include <stdio.h>

int main(void) {
    int age = 0;
    age = 37.2;
    printf("%u", age);
}
```

o compilador gerará um aviso em tempo de compilação e converterá o número decimal em um valor inteiro.

Os tipos de dados internos C são , , , . Vamos descobrir mais sobre eles.`intcharshortlongfloatdoublelong double`

2.1. Números inteiros

C nos fornece os seguintes tipos para definir valores inteiros:

- `char`
- `int`
- `short`
- `long`

Na maioria das vezes, você provavelmente usará `int` para armazenar um inteiro. Mas, em alguns casos, você pode querer escolher uma das outras 3 opções.`int`

O tipo é comumente usado para armazenar letras do gráfico ASCII, mas pode ser usado para manter pequenos inteiros de para . Leva pelo menos 1 byte.`char -128 127`

`int` leva pelo menos 2 bytes. `short` leva pelo menos 2 bytes. `long` leva pelo menos 4 bytes.`short long`

Como você pode ver, não temos a garantia dos mesmos valores para diferentes ambientes. Temos apenas uma indicação. O problema é que os números exatos que podem ser armazenados em cada tipo de dados dependem da implementação e da arquitetura.

Temos a garantia de que não é mais longo do que . E temos a garantia de que não é mais curto do que .`short int long int`

O padrão de especificação ANSI C determina os valores mínimos de cada tipo e, graças a ele, podemos pelo menos saber qual é o valor mínimo que podemos esperar ter à nossa disposição.

Se você estiver programando C em um Arduino, uma placa diferente terá limites diferentes.

Em uma placa Arduino Uno, armazena um valor de 2 bytes, variando de a .
Em um Arduino MKR 1010, armazena um valor de 4 bytes, variando de a .
Uma diferença muito grande.int-32,76832,767int-
2,147,483,6482,147,483,647

Em todas as placas Arduino, armazena um valor de 2 bytes, variando de a .
armazenar 4 bytes, variando de a .short-32,76832,767long-
2,147,483,6482,147,483,647

2.2. Inteiros não assinados

Para todos os tipos de dados acima, podemos preceder para iniciar o intervalo em 0, em vez de um número negativo. Isso pode fazer sentido em muitos casos.`unsigned`

- `unsigned char` irá variar de até pelo menos 0255
- `unsigned int` irá variar de até pelo menos 065,535
- `unsigned short` irá variar de até pelo menos 065,535
- `unsigned long` irá variar de até pelo menos 04,294,967,295

2.3. O problema do estouro

Diante de todos esses limites, uma pergunta pode surgir: como podemos garantir que nossos números não excedam o limite? E o que acontece se excedemos o limite?

Se você tiver um número em 255 e incrementá-lo, receberá 256 em troca. Como esperado. Se você tiver um número em 255 e incrementá-lo, receberá 0 em troca. Ele é redefinido a partir do valor inicial possível.`unsigned int`
`unsigned char`

Se você tiver um número em 255 e adicionar 10 a ele, obterá o número:`unsigned char9`

```
#include <stdio.h>

int main(void) {
    unsigned char j = 255;
    j = j + 10;
    printf("%u", j); /* 9 */
}
```

Se você tiver um valor assinado, o comportamento será indefinido. Ele basicamente lhe dará um grande número que pode variar, como neste caso:

```
#include <stdio.h>

int main(void) {
    char j = 127;
    j = j + 10;
    printf("%u", j); /* 4294967177 */
}
```

Em outras palavras, C não protege você de ultrapassar os limites de um tipo. Você precisa cuidar disso sozinho.

2.4. Advertências ao declarar o tipo errado

Quando você declara a variável e a inicializa com o valor errado, o compilador (aquele que você provavelmente está usando) deve avisá-lo: gcc

```
#include <stdio.h>

int main(void) {
    char j = 1000;
}

hello.c:4:11: warning: implicit conversion
      from 'int' to
```

```
'char' changes value from 1000 to -24
[-Wconstant-conversion]
char j = 1000;
~     ^~~~

1 warning generated.
```

E também avisa em atribuições diretas:

```
#include <stdio.h>

int main(void) {
    char j;
    j = 1000;
}
```

Mas não se você aumentar o número usando, por exemplo:+=

```
#include <stdio.h>

int main(void) {
    char j = 0;
    j += 1000;
}
```

2.5. Números de ponto flutuante

Os tipos de ponto flutuante podem representar um conjunto muito maior de valores do que os inteiros e também podem representar frações, algo que os inteiros não podem fazer.

Usando números de ponto flutuante, representamos números como números decimais vezes potências de 10.

Você pode ver números de ponto flutuante escritos como

- 1.29e-3

- -2.3e+5

e de outras maneiras aparentemente estranhas.

Os seguintes tipos:

- `float`
- `double`
- `long double`

são usados para representar números com pontos decimais (tipos de ponto flutuante). Todos podem representar números positivos e negativos.

Os requisitos mínimos para qualquer implementação C é que pode representar um intervalo entre 10^{-37} e 10^{+37} , e normalmente é implementado usando 32 bits. pode representar um conjunto maior de números. pode conter ainda mais números.`float double long double`

Os números exatos, como acontece com os valores inteiros, dependem da implementação.

Em um Mac moderno, a é representado em 32 bits, e tem uma precisão de 24 bits significativos, 8 bits são usados para codificar o expoente. Um número é representado em 64 bits, com uma precisão de 53 bits significativos, 11 bits são usados para codificar o expoente. O tipo é representado em 80 bits, tem uma precisão de 64 bits significativos, 15 bits são usados para codificar o expoente.`float double long double`

No seu computador específico, como você pode determinar o tamanho específico dos tipos? Você pode escrever um programa para fazer isso:

```
#include <stdio.h>

int main(void) {
    printf("char size: %lu bytes\n", sizeof(char));
    printf("int size: %lu bytes\n", sizeof(int));
    printf("short size: %lu bytes\n", sizeof(short));
```

```
printf("long size: %lu bytes\n", sizeof(long));
printf("float size: %lu bytes\n", sizeof(float));
printf("double size: %lu bytes\n",
       sizeof(double));
printf("long double size: %lu bytes\n",
       sizeof(long double));
}
```

No meu sistema, um Mac moderno, ele imprime:

```
char size: 1 bytes
int size: 4 bytes
short size: 2 bytes
long size: 8 bytes
float size: 4 bytes
double size: 8 bytes
long double size: 16 bytes
```

3. Constantes

Vamos agora falar sobre constantes.

Uma constante é declarada de forma semelhante às variáveis, exceto que ela é precedida pela palavra-chave e você sempre precisa especificar um valor.`const`

Assim:

```
const int age = 37;
```

Isso é perfeitamente válido C, embora seja comum declarar constantes maiúsculas, assim:

```
const int AGE = 37;
```

É apenas uma convenção, mas que pode ajudá-lo muito ao ler ou escrever um programa em C, pois melhora a legibilidade. Nome em maiúsculas significa constante, nome minúsculo significa variável.

Um nome constante segue as mesmas regras para nomes de variáveis: pode conter qualquer letra maiúscula ou minúscula, pode conter dígitos e o caractere de sublinhado, mas não pode começar com um dígito. São nomes de variáveis válidos, não é.`AGEAge101AGE`

Outra maneira de definir constantes é usando esta sintaxe:

```
#define AGE 37
```

Nesse caso, você não precisa adicionar um tipo e também não precisa do sinal de igual e omite o ponto-e-vírgula no final.=

O compilador C inferirá o tipo a partir do valor especificado, em tempo de compilação.

4. Operadores

A C nos oferece uma ampla variedade de operadores que podemos usar para operar com dados.

Em particular, podemos identificar vários grupos de operadores:

- operadores aritméticos
- operadores de comparação
- operadores lógicos
- operadores de atribuição composta
- operadores bit a bit
- operadores de ponteiro
- operadores de estrutura
- operadores diversos

Nesta seção vou detalhar todos eles, usando 2 variáveis imaginárias e como exemplos.ab

Estou mantendo operadores bit a bit, operadores de estrutura e operadores de ponteiro fora desta lista, para manter as coisas mais simples

4.1. Operadores aritméticos

Neste grupo de macros, vou separar operadores binários e operadores unários.

Os operadores binários funcionam usando dois operandos:

Operador	Nome	Exemplo
=	Designação	a = b
+	Adição	a + b
-	Subtração	a - b
*	Multiplicação	a * b
/	Divisão	a / b
%	Módulo	a % b

Os operadores unários só usam um operando:

Operador	Nome	Exemplo
+	Unary mais	+a
-	Unário menos	-a
++	Incremento	a++ ou ++a
--	Decréscimo	a-- ou --a

A diferença entre e é que incrementa a variável depois de usá-la. incrementa a variável antes de usá-la.
a++++aa++a++aa

Por exemplo:

```
int a = 2;
int b;
b = a++ /* b is 2, a is 3 */
b = ++a /* b is 4, a is 4 */
```

O mesmo se aplica ao operador de decréscimo.

4.2. Operadores de comparação

Operador	Nome	Exemplo
<code>==</code>	Operador igual	<code>a == b</code>
<code>!=</code>	Operador não igual	<code>!= b</code>
<code>></code>	Maior que	<code>a > b</code>
<code><</code>	Menos de	<code>a < b</code>
<code>>=</code>	Maior ou igual a	<code>a >= b</code>
<code><=</code>	Menor ou igual a	<code>a <= b</code>

4.3. Operadores lógicos

- `!` NÃO (exemplo: `!a`)
- `&&` E (exemplo: `a && b`)
- `||` OR (exemplo: `a || b`)

Esses operadores são ótimos quando se trabalha com valores booleanos.

4.4. Operadores de atribuição composta

Esses operadores são úteis para executar uma atribuição e, ao mesmo tempo, executar uma operação aritmética:

Operador	Nome	Exemplo
<code>+=</code>	Atribuição de adição	<code>a += b</code>
<code>-=</code>	Atribuição de subtração	<code>a -= b</code>
<code>*=</code>	Atribuição de multiplicação	<code>a *= b</code>
<code>/=</code>	Atribuição de divisão	<code>a /= b</code>
<code>%=</code>	Atribuição de módulo	<code>a %= b</code>

4.5. Operadores diversos

4.5.1. O operador ternário

O operador ternário é o único operador em C que trabalha com 3 operandos, e é um caminho curto para expressar condicionais.

É assim que parece:

```
<condition> ? <expression> : <expression>
```

Exemplo:

```
a ? b : c
```

Se for avaliado como , então a instrução é executada, caso contrário, é.`a truebc`

O operador ternário é funcionalmente igual a um condicional `if/else`, exceto que é mais curto de expressar e pode ser alinhado em uma expressão.

4.5.2. Dimensão

O operador retorna o tamanho do operando que você passa. Você pode passar uma variável, ou até mesmo um tipo.`sizeof`

Exemplo de uso:

```
#include <stdio.h>

int main(void) {
    int age = 37;
    printf("%ld\n", sizeof(age));
    printf("%ld", sizeof(int));
}
```

4.6. Precedência do operador

Com todos esses operadores (e muito mais, que eu não abordei neste post, incluindo operadores de estrutura bit a bit e operadores de ponteiro), devemos prestar atenção ao usá-los juntos em uma única expressão.

Suponha que tenhamos esta operação:

```
int a = 2;
int b = 4;
int c = b + a * a / b - a;
```

Qual é o valor do c ? Obtemos a adição sendo executada antes da multiplicação e da divisão?

Existe um conjunto de regras que nos ajudam a resolver este quebra-cabeça.

Em ordem de menos precedência para mais precedência, temos:

- o operador de atribuição=
- os operadores **binários** e+-
- o e os operadores*/
- os operadores e unários+-

Os operadores também têm uma regra de associatividade, que é sempre da esquerda para a direita, exceto para os operadores unários e a atribuição.

Em:

```
int c = b + a * a / b - a;
```

Primeiro executamos $a * a$, que por ser da esquerda para a direita podemos separar em $a * a$ e o resultado: $2 * 2 = 4$

Então podemos realizar a soma e a subtração: $4 + 1 - 2$. O valor de c é 3

Em todos os casos, no entanto, quero ter certeza de que você percebe que pode usar parênteses para tornar qualquer expressão semelhante mais fácil

de ler e compreender.

Parênteses têm maior prioridade sobre qualquer outra coisa.

A expressão de exemplo acima pode ser reescrita como:

```
int c = b + ((a * a) / b) - a;
```

e não precisamos pensar muito nisso.

5. Condicionais

Qualquer linguagem de programação fornece aos programadores a capacidade de realizar escolhas.

Queremos fazer X em alguns casos, e Y em outros casos.

Queremos verificar os dados e fazer escolhas com base no estado desses dados.

C nos fornece 2 maneiras de fazê-lo.

A primeira é a declaração, com seu ajudante, e a segunda é a declaração `.if .else .switch`

5.1. if

Em uma instrução, você pode verificar se uma condição é verdadeira e, em seguida, executar o bloco fornecido entre colchetes: `if`

```
int a = 1;

if (a == 1) {
    /* do something */
}
```

Você pode acrescentar um bloco para executar um bloco diferente se a condição original for falsa;`else`

```
int a = 1;

if (a == 2) {
    /* do something */
} else {
    /* do something else */
}
```

Cuidado com uma fonte comum de bugs - sempre use o operador de comparação em comparações, e não o operador de atribuição, caso contrário, a verificação condicional sempre será verdadeira, a menos que o argumento seja , por exemplo, se você fizer isso:`==if0`

```
int a = 0;

if (a = 0) {
    /* never invoked */
}
```

Por que isso acontece? Porque a verificação condicional procurará um resultado booleano (o resultado de uma comparação) e o número sempre equivale a um valor falso. Todo o resto é verdade, incluindo números negativos.`0`

Você pode ter vários blocos empilhando várias instruções:`elseif`

```
int a = 1;

if (a == 2) {
    /* do something */
} else if (a == 1) {
    /* do something else */
} else {
```

```
/* do something else again */  
}
```

5.2. switch

Se você precisa fazer muitos se / senão / se blocos para executar uma verificação, talvez porque você precisa verificar o valor exato de uma variável, então pode ser muito útil para você.switch

Você pode fornecer uma variável como condição e uma série de pontos de entrada para cada valor esperado:case

```
int a = 1;  
  
switch (a) {  
    case 0:  
        /* do something */  
        break;  
    case 1:  
        /* do something else */  
        break;  
    case 2:  
        /* do something else */  
        break;  
}
```

Precisamos de uma palavra-chave no final de cada caso, para evitar que o próximo caso seja executado quando o anterior terminar. Esse efeito de "cascata" pode ser útil de algumas maneiras criativas.break

Você pode adicionar um caso "catch-all" no final, rotulado :default

```
int a = 1;  
  
switch (a) {  
    case 0:
```

```

/* do something */
break;

case 1:
    /* do something else */
    break;

case 2:
    /* do something else */
    break;

default:
    /* handle all the other cases */
    break;
}

```

6. Loops

C nos oferece três maneiras de executar um loop: **para loops**, **while loops** e **do while loops**. Todos eles permitem que você itere sobre matrizes, mas com algumas diferenças. Vamos vê-los em detalhes.

6.1. Para loops

A primeira e provavelmente a mais comum maneira de executar um loop é **para loops**.

Usando a palavra-chave, podemos definir as *regras* do loop antecipadamente e, em seguida, fornecer o bloco que será executado repetidamente.`for`

Assim:

```

for (int i = 0; i <= 10; i++) {
    /* instructions to be repeated */
}

```

O bloco contém 3 partes dos detalhes do looping:`(int i = 0; i <= 10; i++)`

- a condição inicial (`int i = 0`)
- o ensaio (`i <= 10`)
- o incremento (`i++`)

Primeiro, definimos uma variável de loop, neste caso chamada `i`. É um nome de variável comum a ser usado para loops, juntamente com `j` para loops aninhados (um loop dentro de outro loop). É apenas uma convenção.

A variável é inicializada no valor 0 e a primeira iteração é feita. Em seguida, ele é incrementado como a parte de incremento diz (neste caso, incrementando por 1), e todo o ciclo se repete até chegar ao número 10.

Dentro do bloco principal do loop podemos acessar a variável para saber em qual iteração estamos. Este programa deve imprimir:

```
i0 1 2 3 4 5 5 6 7 8  
9 10
```

```
for (int i = 0; i <= 10; i++) {  
    /* instructions to be repeated */  
    printf("%u ", i);  
}
```

Os loops também podem começar a partir de um número alto e ir para um número menor, como este:

```
for (int i = 10; i > 0; i--) {  
    /* instructions to be repeated */  
}
```

Você também pode incrementar a variável de loop em 2 ou outro valor:

```
for (int i = 0; i < 1000; i = i + 30) {  
    /* instructions to be repeated */  
}
```

6.2. Enquanto loops

Enquanto loops é mais simples de escrever do que um loop, porque requer um pouco mais de trabalho de sua parte.`for`

Em vez de definir todos os dados do loop antecipadamente quando você inicia o loop, como você faz no loop, usando você apenas verifica se há uma condição:`for``while`

```
while (i < 10) {  
}
```

Isso pressupõe que já está definido e inicializado com um valor.`i`

E esse loop será um loop infinito, a menos que você incremente a variável em algum ponto dentro do **loop**. Um loop infinito é ruim porque bloqueará o programa, nada mais pode acontecer.`i`

Isto é o que você precisa para um loop de enquanto "correto":

```
int i = 0;  
  
while (i < 10) {  
    /* do something */  
  
    i++;  
}
```

Há uma exceção a isso, e veremos isso em um minuto. Antes, deixe-me apresentar `.do while`

6.3. Fazer enquanto loops

Embora os loops sejam ótimos, mas pode haver momentos em que você precise fazer uma coisa específica: você quer sempre executar um bloco e, em seguida, *talvez* repeti-lo.

Isso é feito usando a palavra-chave, de uma maneira muito semelhante a um loop, mas ligeiramente diferente:`do while`

```
int i = 0;

do {
    /* do something */

    i++;
} while (i < 10);
```

O bloco que contém o comentário é sempre executado pelo menos uma vez, independentemente da verificação de condição na parte inferior.`/* do something */`

Então, até que seja menor que 10, repetiremos o bloco.

6.4. Quebrando um loop usando break

Em todos os loops C, temos uma maneira de sair de um loop a qualquer momento, imediatamente, independentemente das condições definidas para o loop.

Isso é feito usando a palavra-chave `break`

Isso é útil em muitos casos. Talvez você queira verificar o valor de uma variável, por exemplo:

```
for (int i = 0; i <= 10; i++) {
    if (i == 4 && someVariable == 10) {
        break;
    }
}
```

Ter essa opção de sair de um loop é particularmente interessante para loops (e também), porque podemos criar loops aparentemente infinitos que

terminam quando uma condição ocorre, e você define isso dentro do bloco de loop:`while`

```
int i = 0;
while (1) {
    /* do something */

    i++;
    if (i == 10) break;
}
```

É bastante comum ter esse tipo de loops em C.

7. Matrizes

Uma matriz é uma variável que armazena vários valores.

Cada valor na matriz, em C, deve ter o **mesmo tipo**. Isso significa que você terá matrizes de valores, matrizes de valores e muito mais.`intdouble`

Você pode definir uma matriz de valores como esta:`int`

```
int prices[5];
```

Você deve sempre especificar o tamanho da matriz. C não fornece matrizes dinâmicas prontas para uso (você precisa usar uma estrutura de dados como uma lista vinculada para isso).

Você pode usar uma constante para definir o tamanho:

```
const int SIZE = 5;
int prices[SIZE];
```

Você pode inicializar uma matriz no momento da definição, desta forma:

```
int prices[5] = { 1, 2, 3, 4, 5 };
```

Mas você também pode atribuir um valor após a definição, desta forma:

```
int prices[5];
```

```
prices[0] = 1;
prices[1] = 2;
prices[2] = 3;
prices[3] = 4;
prices[4] = 5;
```

Ou, mais prático, usando um loop:

```
int prices[5];

for (int i = 0; i < 5; i++) {
    prices[i] = i + 1;
}
```

E você pode fazer referência a um item na matriz usando colchetes após o nome da variável da matriz, adicionando um inteiro para determinar o valor do índice. Assim:

```
prices[0]; /* array item value: 1 */
prices[1]; /* array item value: 2 */
```

Os índices de matriz começam em 0, portanto, uma matriz com 5 itens, como a matriz acima, terá itens que variam de a .pricesprices[0]prices[4]

O interessante sobre matrizes C é que todos os elementos de uma matriz são armazenados sequencialmente, um após o outro. Não é algo que normalmente acontece com linguagens de programação de nível superior.

Outra coisa interessante é esta: o nome da variável da matriz, no exemplo acima, é um ponteiro para o primeiro elemento da matriz e, como tal, pode ser usado como um **ponteiro** normal.`prices`

Mais sobre ponteiros em breve.

8. Cordas

Em C, cadeias de caracteres são um tipo especial de matriz: uma cadeia de caracteres é uma matriz de valores:`char`

```
char name[7];
```

Eu introduzi o tipo quando introduzi tipos, mas em suma, ele é comumente usado para armazenar letras do gráfico ASCII.`char`

Uma cadeia de caracteres pode ser inicializada como você inicializa uma matriz normal:

```
char name[7] = { 'F', 'l', 'a', 'v', 'i', 'o' };
```

Ou, mais convenientemente, com um literal de cadeia de caracteres (também chamado de constante de cadeia de caracteres), uma sequência de caracteres entre aspas duplas:

```
char name[7] = "Flavio";
```

Você pode imprimir uma cadeia de caracteres usando:`printf()`%

```
printf("%s", name);
```

Você percebe como "Flávio" tem 6 caracteres de comprimento, mas eu defini uma matriz de comprimento 7? Por que? Isso ocorre porque o último caractere em uma cadeia de caracteres deve ser um valor, o terminador de cadeia de caracteres, e devemos abrir espaço para ele.`0`

Isso é importante ter em mente, especialmente ao manipular strings.

Falando em manipular cadeias de caracteres, há uma importante biblioteca padrão que é fornecida por C: `.string.h`

Essa biblioteca é essencial porque abstrai muitos dos detalhes de baixo nível do trabalho com cadeias de caracteres e nos fornece um conjunto de funções úteis.

Você pode carregar a biblioteca em seu programa adicionando na parte superior:

```
#include <string.h>
```

E depois de fazer isso, você tem acesso a:

- `strcpy()` para copiar uma cadeia de caracteres sobre outra cadeia de caracteres
- `strcat()` para acrescentar uma cadeia de caracteres a outra cadeia de caracteres
- `strcmp()` para comparar duas cadeias de caracteres para igualdade
- `strncmp()` para comparar os primeiros caracteres de duas cadeias de caracteres
- `strlen()` para calcular o comprimento de uma cadeia de caracteres

e muitos, muitos mais.

9. Ponteiros

Os ponteiros são uma das partes mais confusas / desafiadoras do C, na minha opinião. Especialmente se você é novo em programação, mas também se você vem de uma linguagem de programação de nível superior, como Python ou JavaScript.

Nesta seção, quero apresentá-los da maneira mais simples, mas não emburrada.

Um ponteiro é o endereço de um bloco de memória que contém uma variável.

Quando você declara um número inteiro como este:

```
int age = 37;
```

Podemos usar o operador para obter o valor do endereço na memória de uma variável:&

```
printf("%p", &age); /* 0x7ffef7dcb9c */
```

Usei o formato especificado para imprimir o valor de endereço.%printf()

Podemos atribuir o endereço a uma variável:

```
int *address = &age;
```

Usando na declaração, não estamos declarando uma variável inteira, mas sim um **ponteiro para um inteiro**.int *address

Podemos usar o operador de ponteiro para obter o valor da variável para a qual um endereço está apontando:*

```
int age = 37;
int *address = &age;
printf("%u", *address); /* 37 */
```

Desta vez, estamos usando o operador de ponteiro novamente, mas como não é uma declaração desta vez, significa "o valor da variável para a qual esse ponteiro aponta".

Neste exemplo, declaramos uma variável e usamos um ponteiro para inicializar o valor:age

```
int age;  
int *address = &age;  
*address = 37;  
printf("%u", *address);
```

Ao trabalhar com C, você descobrirá que muitas coisas são construídas em cima desse conceito simples, portanto, certifique-se de se familiarizar um pouco com ele, executando os exemplos acima por conta própria.

Os ponteiros são uma ótima oportunidade porque nos forçam a pensar sobre endereços de memória e como os dados são organizados.

Arrays são um exemplo. Quando você declara uma matriz:

```
int prices[3] = { 5, 4, 3 };
```

A variável é, na verdade, um ponteiro para o primeiro item da matriz. Você pode obter o valor do primeiro item usando esta função neste caso:`prices``printf()`

```
printf("%u", *prices); /* 5 */
```

O legal é que podemos obter o segundo item adicionando 1 ao ponteiro:`prices`

```
printf("%u", *(prices + 1)); /* 4 */
```

E assim por diante para todos os outros valores.

Também podemos fazer muitas operações agradáveis de manipulação de strings, já que strings são arrays sob o capô.

Também temos muito mais aplicativos, incluindo passar a referência de um objeto ou uma função, para evitar consumir mais recursos para copiá-lo.

10. Funções

Funções são a maneira como podemos estruturar nosso código em subrotinas que podemos:

dar um nome a
ligue quando precisarmos deles

A partir do seu primeiro programa, um "Olá, Mundo!", você imediatamente faz uso das funções C:

```
#include <stdio.h>

int main(void) {
    printf("Hello, World!");
}
```

A função é uma função muito importante, pois é o ponto de entrada para um programa C.main()

Aqui está outra função:

```
void doSomething(int value) {
    printf("%u", value);
}
```

As funções têm 4 aspectos importantes:

elas têm um nome, para que possamos invocá-los ("chamá-los") mais tarde
elas especificam um valor de retorno
elas podem ter argumentos
elas têm um corpo, envolto em cintas encaracolados

O corpo da função é o conjunto de instruções que são executadas sempre que invocamos uma função.

Se a função não tiver nenhum valor de retorno, você poderá usar a palavra-chave antes do nome da função. Caso contrário, você especifica o tipo de valor de retorno da função (para um inteiro, para um valor de ponto flutuante, para uma cadeia de caracteres, etc.).

voidintfloatconst char *

Não é possível retornar mais de um valor de uma função.

Uma função pode ter argumentos. Eles são opcionais. Se não os tiver, dentro dos parênteses inserimos, assim:

```
void doSomething(void) {  
    /* ... */  
}
```

Nesse caso, quando invocarmos a função, a chamaremos sem nada entre parênteses:

```
doSomething();
```

Se tivermos um parâmetro, especificamos o tipo e o nome do parâmetro, assim:

```
void doSomething(int value) {  
    /* ... */  
}
```

Quando invocamos a função, passamos esse parâmetro entre parênteses, assim:

```
doSomething(3);
```

Podemos ter vários parâmetros e, em caso afirmativo, os separamos usando uma vírgula, tanto na declaração quanto na invocação:

```
void doSomething(int value1, int value2) {  
    /* ... */  
}  
  
doSomething(3, 4);
```

Os parâmetros são passados por **cópia**. Isso significa que, se você modificar , seu valor será modificado localmente e o valor fora da função, onde foi passado na invocação, não será alterado.value1

Se você passar um **ponteiro** como parâmetro, poderá modificar esse valor de variável porque agora poderá acessá-lo diretamente usando seu endereço de memória.

Não é possível definir um valor padrão para um parâmetro. O C++ pode fazer isso (e, portanto, os programas da Arduino Language podem), mas o C não pode.

Certifique-se de definir a função antes de chamá-la, ou o compilador gerará um aviso e um erro:

```
→ ~ gcc hello.c -o hello; ./hello  
hello.c:13:3: warning: implicit declaration of  
      function 'doSomething' is invalid in C99  
      [-Wimplicit-function-declaration]  
      doSomething(3, 4);  
      ^  
  
hello.c:17:6: error: conflicting types for  
      'doSomething'  
void doSomething(int value1, char value2) {  
      ^  
  
hello.c:13:3: note: previous implicit declaration  
      is here  
      doSomething(3, 4);  
      ^  
  
1 warning and 1 error generated.
```

O aviso que você recebe diz respeito ao pedido, que eu já mencionei.

O erro é sobre outra coisa, relacionada. Como C não "vê" a declaração de função antes da invocação, ele deve fazer suposições. E assume a função de retornar . A função, no entanto, retorna , daí o erro.intvoid

Se você alterar a definição da função para:

```
int doSomething(int value1, int value2) {  
    printf("%d %d\n", value1, value2);  
    return 1;  
}
```

você apenas receberia o aviso, e não o erro:

```
→ ~ gcc hello.c -o hello; ./hello  
hello.c:14:3: warning: implicit declaration of  
      function 'doSomething' is invalid in C99  
      [-Wimplicit-function-declaration]  
doSomething(3, 4);  
^  
1 warning generated.
```

Em qualquer caso, certifique-se de declarar a função antes de usá-la. Mova a função para cima ou adicione o protótipo da função em um arquivo de cabeçalho.

Dentro de uma função, você pode declarar variáveis.

```
void doSomething(int value) {  
    int doubleValue = value * 2;  
}
```

Uma variável é criada no ponto de invocação da função e é destruída quando a função termina e não é visível de fora.

Dentro de uma função, você pode chamar a própria função. Isso é chamado **de recursão** e é algo que oferece oportunidades peculiares.

11. Entrada e Saída

C é uma linguagem pequena, e o "núcleo" de C não inclui nenhuma funcionalidade de Entrada/Saída (E/S).

Isso não é algo exclusivo de C, é claro. É comum que o núcleo da linguagem seja agnóstico em relação à E/S.

No caso de C, a Entrada/Saída é fornecida a nós pela Biblioteca Padrão C através de um conjunto de funções definidas no arquivo de cabeçalho `stdio.h`

Você pode importar essa biblioteca usando:

```
#include <stdio.h>
```

na parte superior do arquivo C.

Esta biblioteca fornece-nos, entre muitas outras funções:

- `printf()`
- `scanf()`
- `sscanf()`
- `fgets()`
- `fprintf()`

Antes de descrever o que essas funções fazem, quero dedicar um minuto para falar sobre **fluxos de E/S**.

Temos 3 tipos de fluxos de E/S em C:

- `stdin` (entrada padrão)
- `stdout` (saída padrão)

- `stderr` (erro padrão)

Com funções de E/S sempre trabalhamos com fluxos. Um fluxo é uma interface de alto nível que pode representar um dispositivo ou um arquivo. Do ponto de vista C, não temos nenhuma diferença na leitura de um arquivo ou na leitura da linha de comando: é um fluxo de E/S em qualquer caso.

Isso é uma coisa a ter em mente.

Algumas funções são projetadas para trabalhar com um fluxo específico, como `printf`, que usamos para imprimir caracteres em `stdout`. Usando sua contraparte mais geral, podemos especificar o fluxo para gravar.
`printf()`
`stdoutf()`

Desde que comecei a falar sobre `printf`, vamos apresentá-lo agora.

11.1. `printf()`

`printf()` é uma das primeiras funções que você usará ao aprender programação em C.

Em sua forma de uso mais simples, você passa uma cadeia de caracteres literal:

```
printf("hey!");
```

e o programa imprimirá o conteúdo da cadeia de caracteres na tela.

Você pode imprimir o valor de uma variável, e é um pouco complicado porque você precisa adicionar um caractere especial, um espaço reservado, que muda dependendo do tipo da variável. Por exemplo, usamos para um dígito inteiro decimal assinado:`%d`

```
int age = 37;
```

```
printf("My age is %d", age);
```

Podemos imprimir mais de uma variável usando vírgulas:

```
int age_yesterday = 36;
int age_today = 37;

printf("Yesterday my age was %d and today is %d", age_yesterday,
age_today);
```

Existem outros especificadores de formato como: %d

- %c para um char
- %s para uma cadeia de caracteres
- %f para números de ponto flutuante
- %p para ponteiros

e muitos mais.

Podemos usar caracteres de escape em , como o que podemos usar para fazer a saída criar uma nova linha. printf() \n

11.2. scanf()

printf() é usado como uma função de saída. Quero introduzir uma função de entrada agora, para que possamos dizer que podemos fazer toda a coisa de E/S: .scanf()

Essa função é usada para obter um valor do usuário que executa o programa, a partir da linha de comando.

Devemos primeiro definir uma variável que manterá o valor que obtemos da entrada:

```
int age;
```

Em seguida, chamamos com 2 argumentos: o formato (tipo) da variável e o endereço da variável: scanf()

```
scanf("%d", &age);
```

Se quisermos obter uma cadeia de caracteres como entrada, lembre-se de que um nome de cadeia de caracteres é um ponteiro para o primeiro caractere, portanto, você não precisa do caractere antes dele:&

```
char name[20];
scanf("%s", name);
```

Aqui está um pequeno programa que usa ambos e :printf()scanf()

```
#include <stdio.h>

int main(void) {
    char name[20];
    printf("Enter your name: ");
    scanf("%s", name);
    printf("you entered %s", name);
}
```

12. Âmbito das variáveis

Quando você define uma variável em um programa C, dependendo de onde você a declara, ela terá um **escopo** diferente.

Isso significa que ele estará disponível em alguns lugares, mas não em outros.

A posição determina 2 tipos de variáveis:

- **variáveis globais**
- **variáveis locais**

Esta é a diferença: uma variável declarada dentro de uma função é uma variável local, como esta:

```
int main(void) {  
    int age = 37;  
}
```

As variáveis locais só são acessíveis de dentro da função e, quando a função termina, elas param sua existência. Eles são limpos da memória (com algumas exceções).

Uma variável definida fora de uma função é uma variável global, como neste exemplo:

```
int age = 37;  
  
int main(void) {  
    /* ... */  
}
```

As variáveis globais são acessíveis a partir de qualquer função do programa, e estão disponíveis para toda a execução do programa, até que ele termine.

Mencionei que as variáveis locais não estão mais disponíveis após o término da função.

O motivo é que as variáveis locais são declaradas na **pilha**, por padrão, a menos que você as aloque explicitamente no heap usando ponteiros, mas então você precisa gerenciar a memória por conta própria.

13. Variáveis estáticas

Dentro de uma função, você pode inicializar uma **variável estática** usando a palavra-chave `static`

Eu disse "dentro de uma função", porque as variáveis globais são estáticas por padrão, então não há necessidade de adicionar a palavra-chave.

O que é uma variável estática? Uma variável estática é inicializada como 0 se nenhum valor inicial for especificado e retém o valor entre as chamadas de função.

Considere esta função:

```
int incrementAge() {  
    int age = 0;  
    age++;  
    return age;  
}
```

Se ligarmos uma vez, obteremos como valor de retorno. Se chamarmos isso mais de uma vez, sempre obteremos 1 de volta, porque é uma variável local e é reinicializada em cada chamada de função.
incrementAge() 1
age 0

Se alterarmos a função para:

```
int incrementAge() {  
    static int age = 0;  
    age++;  
    return age;  
}
```

Agora, toda vez que chamarmos essa função, obteremos um valor incrementado:

```
printf("%d\n", incrementAge());  
printf("%d\n", incrementAge());  
printf("%d\n", incrementAge());
```

nos dará

1
2

Também podemos omitir a inicialização para 0 em , e apenas gravar porque as variáveis estáticas são automaticamente definidas como 0 quando criadas.
age static int age = 0; static int age;

Também podemos ter matrizes estáticas. Nesse caso, cada item único na matriz é inicializado para 0:

```
int incrementAge() {  
    static int ages[3];  
    ages[0]++;  
    return ages[0];  
}
```

14. Variáveis globais

Nesta seção, quero mencionar a diferença entre **variáveis globais e locais**.

Uma **variável local** é definida dentro de uma função e só está disponível dentro dessa função.

Assim:

```
#include <stdio.h>  
  
int main(void) {  
    char j = 0;  
    j += 10;  
    printf("%u", j); //10  
}
```

j não está disponível em nenhum lugar fora da função.main

Uma **variável global** é definida fora de qualquer função, como esta:

```
#include <stdio.h>

char i = 0;

int main(void) {
    i += 10;
    printf("%u", i); //10
}
```

Uma variável global pode ser acessada por qualquer função no programa. O acesso não se limita à leitura do valor: a variável pode ser atualizada por qualquer função.

Devido a isso, as variáveis globais são uma maneira que temos de compartilhar os mesmos dados entre as funções.

A principal diferença com as variáveis locais é que a memória alocada para as variáveis é liberada quando a função termina.

As variáveis globais só são liberadas quando o programa termina.

15. Definições de tipo

A palavra-chave em C permite que você defina novos tipos.`typedef`

A partir dos tipos C internos, podemos criar nossos próprios tipos, usando esta sintaxe:

```
typedef existingtype NEWTYPE;
```

O novo tipo que criamos é geralmente, por convenção, maiúsculo.

Isso para distingui-lo mais facilmente e reconhecê-lo imediatamente como tipo.

Por exemplo, podemos definir um novo tipo que é um :NUMBERint

```
typedef int NUMBER;
```

e depois de fazer isso, você pode definir novas variáveis:NUMBER

```
NUMBER one = 1;
```

Agora você pode perguntar: por quê? Por que não usar apenas o tipo embutido em vez disso?int

Bem, fica realmente útil quando emparelhado com duas coisas: tipos enumerados e estruturas.typedef

16. Tipos enumerados

Usando as palavras-chave e podemos definir um tipo que pode ter um valor ou outro.typefenum

É um dos usos mais importantes da palavra-chave.typedef

Esta é a sintaxe de um tipo enumerado:

```
typedef enum {
    //...values
} TYPENAME;
```

O tipo enumerado que criamos é geralmente, por convenção, maiúsculo.

Aqui está um exemplo simples:

```
typedef enum {
    true,
    false
} BOOLEAN;
```

C vem com um tipo, então este exemplo não é realmente prático, mas você entendeu a ideia.`bool`

Outro exemplo é definir dias úteis:

```
typedef enum {  
    monday,  
    tuesday,  
    wednesday,  
    thursday,  
    friday,  
    saturday,  
    sunday  
} WEEKDAY;
```

Aqui está um programa simples que usa esse tipo enumerado:

```
#include <stdio.h>  
  
typedef enum {  
    monday,  
    tuesday,  
    wednesday,  
    thursday,  
    friday,  
    saturday,  
    sunday  
} WEEKDAY;  
  
int main(void) {  
    WEEKDAY day = monday;  
  
    if (day == monday) {  
        printf("It's monday!");  
    } else {  
        printf("It's not monday");  
    }  
}
```

```
 }  
 }
```

Cada item na definição de enum é emparelhado a um inteiro, internamente. Então, neste exemplo é 0, é 1 e assim por diante.

Isso significa que o condicional poderia ter sido em vez de , mas é muito mais simples para nós, humanos, raciocinar com nomes em vez de números, por isso é uma sintaxe muito conveniente.

```
if (day == 0)if (day ==  
monday)
```

17. Estruturas

Usando a palavra-chave, podemos criar estruturas de dados complexas usando tipos C básicos.

struct

Uma estrutura é uma coleção de valores de diferentes tipos. Arrays em C são limitados a um tipo, de modo que as estruturas podem revelar-se muito interessantes em muitos casos de uso.

Esta é a sintaxe de uma estrutura:

```
struct <structname> {  
    //...variables  
};
```

Exemplo:

```
struct person {  
    int age;  
    char *name;  
};
```

Você pode declarar variáveis que têm como tipo essa estrutura adicionando-as após o colchete de fechamento, antes do ponto-e-vírgula, desta forma:

```
struct person {
    int age;
    char *name;
} flavio;
```

Ou vários, como este:

```
struct person {
    int age;
    char *name;
} flavio, people[20];
```

Nesse caso, declaro uma única variável chamada , e uma matriz de 20 chamada .personflaviopersonpeople

Também podemos declarar variáveis mais tarde, usando esta sintaxe:

```
struct person {
    int age;
    char *name;
};

struct person flavio;
```

Podemos inicializar uma estrutura no momento da declaração:

```
struct person {
    int age;
    char *name;
};

struct person flavio = { 37, "Flavio" };
```

e uma vez que temos uma estrutura definida, podemos acessar os valores nela usando um ponto:

```
struct person {
    int age;
    char *name;
};

struct person flavio = { 37, "Flavio" };
printf("%s, age %u", flavio.name, flavio.age);
```

Também podemos alterar os valores usando a sintaxe de pontos:

```
struct person {
    int age;
    char *name;
};

struct person flavio = { 37, "Flavio" };

flavio.age = 38;
```

As estruturas são muito úteis porque podemos passá-las como parâmetros de função, ou valores de retorno, incorporando várias variáveis dentro delas, e cada variável tem um rótulo.

É importante notar que as estruturas são **passadas por cópia**, a menos que, é claro, você passe um ponteiro para uma struct, caso em que ela é passada por referência.

Usando podemos simplificar o código ao trabalhar com estruturas.`typedef`

Vamos dar um exemplo:

```
typedef struct {
    int age;
    char *name;
} PERSON;
```

A estrutura que criamos usando é geralmente, por convenção, maiúscula.`typedef`

Agora podemos declarar novas variáveis como esta:`PERSON`

```
PERSON flavio;
```

e podemos inicializá-los na declaração desta forma:

```
PERSON flavio = { 37, "Flavio" };
```

18. Parâmetros de linha de comando

Em seus programas C, você pode ter a necessidade de aceitar parâmetros da linha de comando quando o comando é iniciado.

Para necessidades simples, tudo o que você precisa fazer é alterar a assinatura da função `demain()`

```
int main(void)
```

Para

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

`argc` é um número inteiro que contém o número de parâmetros que foram fornecidos na linha de comando.

`argv` é uma matriz de cadeias de caracteres.

Quando o programa é iniciado, recebemos os argumentos nesses 2 parâmetros.

Observe que sempre há pelo menos um item na matriz: o nome do programa `argv`

Tomemos o exemplo do compilador C que usamos para executar nossos programas, assim:

```
gcc hello.c -o hello
```

Se este fosse o nosso programa, teríamos sido 4 e sendo uma matriz contendo argc argv

- gcc
- hello.c
- -o
- hello

Vamos escrever um programa que imprima os argumentos que recebe:

```
#include <stdio.h>

int main (int argc, char *argv[]) {
    for (int i = 0; i < argc; i++) {
        printf("%s\n", argv[i]);
    }
}
```

Se o nome do nosso programa é e nós o executamos assim: , nós teríamos isso como saída:hello./hello

```
./hello
```

Se passarmos alguns parâmetros aleatórios, como este: teríamos essa saída para o terminal:./hello a b c

```
./hello
a
b
c
```

Este sistema funciona muito bem para necessidades simples. Para necessidades mais complexas, existem pacotes comumente usados, como o **getopt**.

19. Arquivos de cabeçalho

Programas simples podem ser colocados em um único arquivo, mas quando seu programa cresce, é impossível manter tudo em apenas um arquivo.

Você pode mover partes de um programa para um arquivo separado e, em seguida, criar um **arquivo de cabeçalho**.

Um arquivo de cabeçalho se parece com um arquivo C normal, exceto que termina com `#include <stdio.h>` vez de `#include <stdio.h>`, e em vez das implementações de suas funções e das outras partes de um programa, ele contém as **declarações**.

Você já usou arquivos de cabeçalho quando usou a função pela primeira vez ou outra função de E/S e teve que digitar:`printf()`

```
#include <stdio.h>
```

para usá-lo.

`#include` é uma diretiva de pré-processador.

O pré-processador vai e procura o arquivo na biblioteca padrão, porque você usou colchetes em torno dele. Para incluir seus próprios arquivos de cabeçalho, você usará aspas, como esta:`stdio.h`

```
#include "myfile.h"
```

O acima será procurado na pasta atual.`myfile.h`

Você também pode usar uma estrutura de pastas para bibliotecas:

```
#include "myfolder/myfile.h"
```

Vamos dar um exemplo. Este programa calcula os anos desde um determinado ano:

```
#include <stdio.h>

int calculateAge(int year) {
    const int CURRENT_YEAR = 2020;
    return CURRENT_YEAR - year;
}

int main(void) {
    printf("%u", calculateAge(1983));
}
```

Suponha que eu queira mover a função para um arquivo separado:calculateAge

Eu crio um arquivo:calculate_age.c

```
int calculateAge(int year) {
    const int CURRENT_YEAR = 2020;
    return CURRENT_YEAR - year;
}
```

E um arquivo onde eu coloco o *protótipo da função*, que é o mesmo que a função no arquivo, exceto o corpo:calculate_age.h.c

```
int calculateAge(int year);
```

Agora no arquivo principal, podemos ir e remover a definição da função, e podemos importar, o que tornará a função disponível: .ccalculateAge()calculate_age.hcalculateAge()

```
#include <stdio.h>
#include "calculate_age.h"

int main(void) {
    printf("%u", calculateAge(1983));
}
```

Não se esqueça de que, para compilar um programa composto por vários arquivos, você precisa listá-los todos na linha de comando, assim:

```
gcc -o main main.c calculate_age.c
```

E com configurações mais complexas, um Makefile é necessário para dizer ao compilador como compilar o programa.

20. O pré-processador

O pré-processador é uma ferramenta que nos ajuda muito na hora de programar com C. Faz parte do C Standard, assim como a linguagem, o compilador e a biblioteca padrão.

Ele analisa nosso programa e garante que o compilador obtenha todas as coisas necessárias antes de prosseguir com o processo.

O que faz, na prática?

Por exemplo, ele procura todos os arquivos de cabeçalho que você inclui com a diretiva `#include`

Ele também analisa cada constante que você definiu usando e a substitui por seu valor real. `#define`

Isso é apenas o começo, e eu mencionei essas 2 operações porque elas são as mais comuns. O pré-processador pode fazer muito mais.

Você notou e teve um no começo? Isso é comum a todas as diretivas de pré-processador. Se uma linha começar com `#`, isso é cuidado pelo pré-processador.`#include#define##`

20.1. Condicionais

Uma das coisas que podemos fazer é usar condicionais para alterar a forma como nosso programa será compilado, dependendo do valor de uma expressão.

Por exemplo, podemos verificar se a constante é 0:`DEBUG`

```
#include <stdio.h>

const int DEBUG = 0;

int main(void) {
    #if DEBUG == 0
        printf("I am NOT debugging\n");
    #else
        printf("I am debugging\n");
    #endif
}
```

20.2. Constantes simbólicas

Podemos definir uma **constante simbólica**:

```
#define VALUE 1
#define PI 3.14
#define NAME "Flavio"
```

Quando usamos `NAME` ou `PI` ou `VALUE` em nosso programa, o pré-processador substitui seu nome pelo valor, antes de executar o programa.

Constantes simbólicas são muito úteis porque podemos dar nomes a valores sem criar variáveis em tempo de compilação.

20.3. Macros

Com também podemos definir uma **macro**. A diferença entre uma macro e uma constante simbólica é que uma macro pode aceitar um argumento e normalmente contém código, enquanto uma constante simbólica é um valor:`#define`

```
#define POWER(x) ((x) * (x))
```

Observe os parênteses ao redor dos argumentos, uma boa prática para evitar problemas quando a macro é substituída no processo de pré-compilação.

Então podemos usá-lo em nosso código assim:

```
printf("%u\n", POWER(4)); //16
```

A grande diferença com as funções é que as macros não especificam o tipo de seus argumentos ou valores de retorno, o que pode ser útil em alguns casos.

20.4. Se definido

Podemos verificar se uma constante simbólica ou uma macro é definida usando:`#ifdef`

```
#include <stdio.h>
#define VALUE 1

int main(void) {
#ifndef VALUE
    printf("Value is defined\n");
}
```

```
#else
    printf("Value is not defined\n");
#endif
}
```

Também temos que verificar o oposto (macro não definida).#ifndef

Também podemos usar e fazer a mesma tarefa.#if defined#ifndef

É comum envolver algum bloco de código em um bloco como este:

```
#if 0
```

```
#endif
```

para impedir temporariamente que ele seja executado ou para usar uma constante simbólica DEBUG:

```
#define DEBUG 0
```

```
#if DEBUG
```

```
    //code only sent to the compiler
    //if DEBUG is not 0
```

```
#endif
```

20.5. Constantes simbólicas predefinidas que você pode usar

O pré-processador também define uma série de constantes simbólicas que você pode usar, identificadas pelos 2 sublinhados antes e depois do nome, incluindo:

- __LINE__ converte para a linha atual no arquivo de código-fonte
- __FILE__ converte para o nome do arquivo
- __DATE__ traduz para a data de compilação, no formato Mmm gg aaaa

- _TIME_ traduz para o tempo de compilação, no formato hh:mm:ss



→ Você pode me seguir no [Twitter](#)



→ Todos os anos organizo um [BOOTCAMP](#) de codificação para lhe ensinar JS, Tailwind, React e Next.js (próxima edição Q1 2024)



→ Quer aprender JavaScript AGORA MESMO com um currículo bem organizado? [O CURSO JS](#) é o lugar



→ Vá para [o SOLO LAB](#) se você estiver em iniciar e administrar um negócio na Internet como uma pessoa solo (NOVO CURSO EM BREVE, junte-se à lista de espera para ficar por dentro)



→ Leia meus outros ebooks [O Manual C](#) [O Manual da Linha de Comando](#) [O Manual CSS](#) [O Manual do Expresso](#) [O Manual do Go](#) [O Manual HTML](#) [O Manual do JS](#) [O Próximo.js](#) [Manual O Nó.js](#) [Manual O Manual do PHP](#) [O Manual do Python](#) [O Manual do React](#)

[O Manual do SQL](#)

[O Manual](#)

[do Svelte](#)

[O Manual Swift](#)