

Revisão da linguagem C

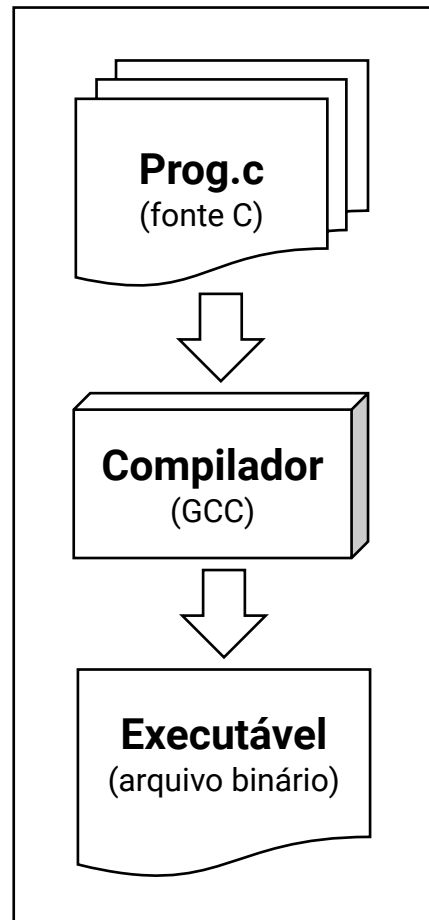
Projeto de arquivos
Prof. Allan Rodrigo Leite

Linguagens de programação

- As linguagens de programação de alto nível impulsionam o desenvolvimento de software desde o surgimento dos computadores
 - É uma abstração para instruções complexas em linguagem de máquina
 - Primeira linguagem de programação surgiu na década de 50
- Instruções em linguagens de programação de alto nível são escritas de forma muito mais clara e legível para o desenvolvedor

Compilador

- Como o computador compreende um programa desenvolvido em linguagem de alto nível?
 - Requer um processo de tradução da linguagem de alto nível para linguagem de montagem
 - Este processo de tradução é conhecido por compilação
- Mesmo assim, o programador precisa seguir uma série de regras ao utilizar uma linguagem de alto nível



```
void swap(int v[], int k) {  
    int aux;  
    temp = v[k];  
    v[k] = v[k + 1];  
    v[k + 1] = aux;  
}
```

Programa em linguagem de
alto nível (C)

Compilador

```
swap:  
    muli    $2, $5, 2  
    add     $2, $5, 4  
    lw      $15, 0($2)  
    lw      $16, 4($2)  
    sw      $16, 0($2)  
    sw      $15, 4($2)  
    jr      $31
```

Programa em
assembly (MIPS)

Montador

```
0000000001010000100000000000011000  
000000000000110000001100000100001  
10001100011000100000000000000000  
100011001111001000000000000000100  
10101100111100100000000000000000  
101011000110001000000000000000100  
000000111110000000000000000001000
```

Programa binário em linguagem de
máquina (MIPS)

Unidades computacionais

- BIT (BInary DiGiT) – dígito binário
 - Menor unidade de informação
 - Armazena somente um valor 0 ou 1
- Byte (BinarY TErm) – termo binário
 - Conjunto de 8 bits
 - Pode representar números, letras, símbolos, imagens, etc.
- Palavra (Word)
 - É a quantidade de bits que a CPU pode processar por vez
 - Atualmente são comuns palavras de 32 ou 64 bits

Variáveis e constantes

- Tipos de dados

Tipo	Tamanho	Menor valor	Maior valor
char	1 byte	-128	+127
unsigned char	1 byte	0	+255
short int (short)	2 bytes	-32.768	+32.767
unsigned short int	2 bytes	0	+65.535
int (*)	4 bytes	-2.147.483.648	+2.147.483.647
long int (long)	4 bytes	-2.147.483.648	+2.147.483.647
unsigned long int	4 bytes	0	+4.294.967.295
float	4 bytes	-10^{38}	$+10^{38}$
double	8 bytes	-10^{308}	$+10^{308}$

(*) depende da máquina, sendo 4 bytes para arquiteturas de 32 bits

Variáveis e constantes

- Constante
 - Valor armazenado na memória
 - Possui um tipo, indicado pela sintaxe

123 /*constante inteira do tipo int*/

12.45 /*constante real do tipo double*/

1245e-2 /*constante real do tipo double*/

12.45F /*constante real do tipo float*/

Variáveis e constantes

- Variável
 - Espaço na memória para armazenar um dado
 - Não é uma variável no sentido matemático
- Uma variável possui
 - Identificador
 - Nome exclusivo para identificar e acessar o espaço de memória
 - Tipo
 - Define a natureza do dado
 - Escopo
 - Determina onde (local) a variável pode ser acessada

Variáveis e constantes


- Declaração de variável
 - Devem ser explicitamente declaradas
 - Podem ser declaradas em conjunto
 - Somente armazenam valores do mesmo tipo com que foram declaradas

```
char a;    /*declara uma variável do tipo char*/  
int b;     /*declara uma variável do tipo int*/  
float c;   /*declara uma variável do tipo float*/  
int d, e;  /*declara duas variáveis do tipo int*/
```

Variáveis e constantes

- Quando uma variável é declarada, seu valor inicial não é modificado e seu conteúdo é desconhecido
 - Comumente estes valores iniciais desconhecidos são chamados de lixo

```
int main() {  
    int a;  
    int b;  
    b = a;  
    a = 10;  
    return 0;  
}
```




	0	1	2	3	4	5	6	7
...	0	1	1	1	0	0	1	0
1712	1	1	0	0	1	1	1	1
a	0	0	0	0	1	1	0	0
	0	1	1	0	0	1	0	1
	1	0	0	1	0	0	0	0
	0	0	0	0	1	0	1	0
1717	1	1	1	1	1	0	1	1
...	1	0	0	1	0	1	0	0

Variáveis e constantes

- Para evitar problemas, recomenda-se que nenhuma variável deve ser utilizada antes de ser inicializada
 - b recebe 1100011001011001000000001010 (em decimal, 207982602)
 - Em outra execução, o valor pode ser outro

```
int main() {  
    int a;  
    int b;  
    b = a;  
    a = 10;  
    return 0;  
}
```



A large arrow points from the variable 'a' in the code to the memory layout table.

	0	1	2	3	4	5	6	7
...	0	1	1	1	0	0	1	0
1712	1	1	0	0	1	1	1	1
a	0	0	0	0	1	1	0	0
	0	1	1	0	0	1	0	1
	1	0	0	1	0	0	0	0
	0	0	0	0	1	0	1	0
1717	1	1	1	1	1	0	1	1
...	1	0	0	1	0	1	0	0

Expressões

- Combinação de variáveis, constantes e operadores que, quando avaliada, produz algum valor
- Tipos de expressões
 - Atribuição
 - Variável recebe um determinado valor
 - Expressão aritmética, incremento ou decremento
 - Resulta em um número (inteiro ou real)
 - Expressão lógica ou relacional
 - Resulta em VERDADEIRO ou FALSO

Expressões aritméticas

Tipo	Operador	Descrição	Inteiros	Reais
Unário	–	Sinal negativo	–2	–2.0
			–a	–b
Binário	+	Adição	a + 2	b + 2.0
	–	Subtração	a – 2	b – 2.0
	*	Multiplicação	a * 2	b * 2.0
	/	Divisão	a / 2	b / 2.0
	%	Módulo	a % 2	Operação não definida para reais

Operadores e expressões

- Operadores de atribuição ($=$, $+=$, $-=$, $*=$, $/=$, $\%=$)
 - A atribuição é tratada como uma expressão
 - A ordem é da direita para a esquerda
 - Também oferece uma notação compacta para atribuições em que a mesma variável aparece nos dois lados

```
i += 2;      //é equivalente a i = i + 2  
x *= y + 1;  //é equivalente a x = x * (y + 1)
```

Operadores e expressões

- Operadores de incremento e decremento (++ , --)
 - Incrementa ou decrementa uma unidade de valor de uma variável
 - Estes operadores não se aplicam a expressão
 - O incremento/decremento pode ocorrer antes ou depois do uso da variável

`n++; //incrementa uma unidade em n, depois de ser usado`

`++n; //incrementa uma unidade em n, antes de ser usado`

`n = 5;`

`x = n++;`

`x = ++n;`

`a = 3;`

`b = a++ * 2;`

Operadores e expressões

- Operadores de incremento e decremento (++ , --)
 - Incrementa ou decrementa uma unidade de valor de uma variável
 - Estes operadores não se aplicam a expressão
 - O incremento/decremento pode ocorrer antes ou depois do uso da variável

`n++; //incrementa uma unidade em n, depois de ser usado`

`++n; //incrementa uma unidade em n, antes de ser usado`

`n = 5;`

`x = n++; //x recebe 5 e n é incrementado para 6`

`x = ++n; //n é incrementado para 7 e x recebe 7`

`a = 3;`

`b = a++ * 2; //a é incrementado para 4 e b recebe 6`

Operadores e expressões

- Operadores relacionais (<, <=, ==, >=, >, !=)
 - O resultado será 0 ou 1
 - Equivalente a FALSO (igual a 0) ou VERDADEIRO (diferente de 0)

```
int a, b;  
int c = 23;  
int d = c + 4;
```

```
c < 20  
d > c
```

Operadores e expressões

- Operadores relacionais (<, <=, ==, >=, >, !=)
 - O resultado será 0 ou 1
 - Equivalente a FALSO (igual a 0) ou VERDADEIRO (diferente de 0)

```
int a, b;  
int c = 23;  
int d = c + 4;
```

```
c < 20 //retorna 0  
d > c  //retorna 1
```

Operadores e expressões

- Operadores lógicos (&&, ||, !)
 - A avaliação ocorre da esquerda para a direita
 - A avaliação para quando o resultado for conhecido, antes mesmo de completar a expressão

```
int a, b;  
int c = 23;  
int d = c + 4;
```

```
a = (c < 20) || (d > c);  
b = (c < 20) && (d > c);
```

Operadores e expressões

- Operadores lógicos (&&, ||, !)
 - A avaliação ocorre da esquerda para a direita
 - A avaliação para quando o resultado for conhecido, antes mesmo de completar a expressão

```
int a, b;  
int c = 23;  
int d = c + 4;
```

```
a = (c < 20) || (d > c); //1 e as duas expressões são validadas  
b = (c < 20) && (d > c); //0 e só a primeira expressão é validada
```

Operadores e expressões

- Função `sizeof`
 - Retorna o número de bytes ocupados por um tipo de dados

```
int a = sizeof(float); //armazena 4 em a
```

- Conversão de tipo de dados
 - Conversão de tipo é automática na avaliação de uma expressão
 - Conversão de tipo pode ser requisitada explicitamente

```
float f;  
float f = 3;           //f recebe 3.0F, conversão implícita  
int g, h;  
g = (int) 3.5;         //3.5 é convertido (e arredondado) para int  
h = (int) 3.5 % 2;     //conversão realizada antes do módulo
```

Entrada e saída

- Função `printf`
 - Possibilita a saída de valores conforme o formato especificado

```
printf(formato, expr1, expr2, ..., exprN);
```

```
printf("%d %g", 33, 5.3);  
//imprimirá na console a linha "33 5.3"
```

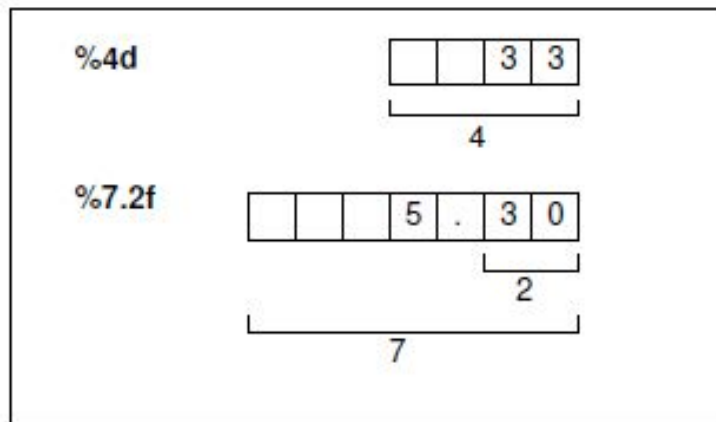
```
printf("Inteiro = %d Real = %g", 33, 5.3);  
//imprimirá na console a linha "Inteiro = 33 Real = 5.3"
```

Entrada e saída

- Formatos do `printf`
 - `%c` especifica um `char`
 - `%d` especifica um `int`
 - `%u` especifica um `unsigned int`
 - `%f` especifica um `double` ou `float`
 - `%e` especifica um `double` ou `float` em formato científico
 - `%g` especifica um `double` ou `float` em formato mais apropriado (`%f` ou `%e`)
 - `%s` especifica uma cadeia de caracteres
- Caractere de escape
 - `\n` quebra de linha
 - `\t` tabulação
 - `\"` caractere `"`
 - `\\` caractere `\`

Entrada e saída

- Tamanhos de campo printf



Entrada e saída

- Função scanf
 - Captura valores fornecidos via teclado

```
scanf(formato, var1, var2, ..., varN);
```

```
int n;
```

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

```
//valor inteiro digitado pelo usuário é armazenado em n
```

Entrada e saída

- Formatos do scanf
 - %c especifica um char
 - %d especifica um int
 - %u especifica um unsigned int
 - %f, %e, %g especifica um float
 - %lf, %le, %lg especifica um double
 - %s especifica uma cadeia de caracteres
- Caracteres diferentes dos especificadores
 - Servem para cercar a entrada

```
scanf("%d:%d", &hora, &minuto);
```

Controle de fluxo

- Instrução `if`
 - Comando básico para definir desvios ou tomada de decisão
 - Se condição `<expr>` for verdadeira (diferente de 0), executa `<bloco 1>`
 - Se condição `<expr>` for falsa (igual a 0), executa o `<bloco 2>`

```
if (<expr>) {  
    <bloco 1>  
}  
[ else {  
    <bloco 2>  
} ]
```

Controle de fluxo

- Estrutura de bloco
 - Declaração de variáveis
 - Só podem ocorrer no início do corpo da função ou bloco
 - Esta restrição não existe em versões mais recentes do C (C99)
 - Escopo de uma variável
 - Uma variável declarada dentro de um bloco é válida apenas no próprio bloco
 - Após o término da execução do bloco, a variável deixa de existir

```
if (n > 0) {  
    int i;  
    ...  
}  
//a variável i não existe mais
```

Controle de fluxo

- Instrução switch
 - Seleciona um bloco entre várias opções

```
switch (<expr>) {  
    case <opção 1>:  
        <bloco 1>;  
        break;  
    case <opção 2>:  
        <bloco 2>;  
        break;  
    default:  
        <bloco padrão>;  
        break;  
}
```

Laços de repetição

- Fatorial de número inteiro não negativo
 - $n! = n \times (n - 1) \times (n - 2) \times \dots$
- Cálculo não recursivo de `fatorial(n)`
 - Comece com $k = 1$ e $f = 1$
 - Faça enquanto $k \leq n$
 - f recebe $f * k$
 - Incrementa k em uma unidade

Laços de repetição

- Instrução `while`
 - Enquanto `<expr>` for verdadeira, o `<bloco>` é executado
 - Quando `<expr>` for falsa, o laço de repetição é encerrado

```
while (<expr>) {  
    <bloco de instruções>  
}
```

Laços de repetição

```
int main () {  
    int k, n;  
    int f = 1;  
    printf("Digite um numero inteiro nao negativo:");  
    scanf("%d", &n);  
    k = 1;  
    while (k <= n) {  
        f = f * k; /* f = f * k é equivalente a f *= k */  
        k = k + 1; /* k = k + 1 é equivalente a k++ */  
    }  
    printf("Fatorial = %d \n", f);  
    return 0;  
}
```


Laços de repetição

- Instrução for
 - Forma compacta para definir laços
 - Expressão inicial
 - Condição de parada
 - Expressão de incremento

```
for (<expr inicial>; <condição>; <expr incremento>) {  
    <bloco de instruções>  
}
```

Laços de repetição

```
int main () {  
    int k, n;  
    int f = 1;  
    printf("Digite um numero inteiro nao negativo:");  
    scanf("%d", &n);  
    for (k = 1; k <= n; k++) {  
        f *= k;  
    }  
    printf(" Fatorial = %d \n", f);  
    return 0;  
}
```

Laços de repetição

- Comando `do while`
 - Condição de parada é avaliada ao final do bloco
 - Portanto, sempre executará uma vez o bloco de repetição

```
do {  
    <bloco de instruções>  
} while (<condição>);
```

Laços de repetição

```
int main () {  
    int k, n;  
    int f = 1;  
    do {  
        printf("Digite um numero inteiro nao negativo:");  
        scanf("%d", &n);  
    } while (n < 0);  
    for (k = 1; k <= n; k++) {  
        f *= k;  
    }  
    printf(" Fatorial = %d \n", f);  
    return 0;  
}
```

Laços de repetição

- Comandos break e continue
 - break termina o laço de repetição
 - continue termina a iteração atual e vai para a próxima

```
for (int i = 0; i < 10; i++) {  
    if (i == 5) break;  
    printf("%d ", i);  
}
```

```
for (int i = 0; i < 10; i++) {  
    if (i == 5) continue;  
    printf("%d ", i);  
}
```

Variáveis e ponteiros

- Variáveis comuns: possuem um endereço de memória predefinido
- Ponteiros: apontam para um endereço de memória
- Operadores unários &, * e **
 - Operador &
 - Endereço de ...
 - Utilizado para retornar o endereço de memória de uma variável
 - Operador *
 - Conteúdo de ...
 - Utilizado para indicar o endereço de memória apontado
 - Operador **
 - Ponteiro de ponteiro

Variáveis e ponteiros

- Declaração de ponteiros

<tipo de dado>*

int* var;

<tipo de dado>**

int** var;

- Vetores são essencialmente ponteiros
 - O acesso aos índices é chamado de aritmética de ponteiros

Variáveis e ponteiros

- Aritmética de ponteiros

```
int main() {  
    int v[5] = { 1, 2, 3, 4, 5 };  
    printf("%p\n", v);  
    printf("%d\n", v[0]);  
    printf("%d\n", *v);  
    printf("%p\n", v + 3);  
    printf("%d\n", *(v + 3));  
}
```


Variáveis e ponteiros

- Aritmética de ponteiros

```
int main() {  
    int v[5] = { 1, 2, 3, 4, 5 };  
    printf("%p\n", v);           //endereço de v  
    printf("%d\n", v[0]);        //valor do primeiro elemento de v (1)  
    printf("%d\n", *v);          //valor do primeiro elemento de v (1)  
    printf("%p\n", v + 3);       //endereço do quarto elemento de v  
    printf("%d\n", *(v + 3));    //valor no quarto elemento de v (4)  
}
```

Alocação dinâmica de memória

- É possível reservar espaços de memória em tempo de execução
- Maneiras para manipular espaços de memória
 - Variáveis globais e estáticas
 - O espaço reservado para uma variável global existe enquanto o programa estiver sendo executado
 - Variáveis locais
 - O espaço existe enquanto a função que declarou a variável está em execução
 - Isto é, o espaço é liberado para outro uso após o fim da execução da função
 - Reservar memória em tempo de execução
 - Solicita-se ao SO um espaço de um determinado tamanho

Alocação dinâmica de memória

- Função malloc(size_t n)
 - Aloca dinamicamente um espaço

```
int *a = malloc(sizeof(int)); //aloca dinamicamente 4 bytes - int
*a = 10;
```

- Criando vetores dinâmicos

```
int *a = malloc(3 * sizeof(int)); //aloca um vetor 4 bytes em cada índice
a[0] = 10;
a[1] = 20;
a[2] = 30;
```

Alocação dinâmica de memória

- Função `calloc(size_t n, size_t size)`
 - Aloca `n` espaços contínuos de memória com `size` tamanho
 - O `calloc` inicializa a memória alocada com zero
 - O `malloc` não realiza nenhum tipo de inicialização
- Criando vetores dinâmicos

```
int *a = calloc(3, sizeof(int)); //aloca um vetor 4 bytes em cada índice
a[0] = 10;
a[1] = 20;
a[2] = 30;
```

Alocação dinâmica de memória

- Função `free(void *p)`
 - Desaloca a memória ocupada por um ponteiro
- Criando vetores dinâmicos

```
int *a = calloc(3, sizeof(int)); //aloca um vetor 4 bytes em cada índice
a[0] = 10;
a[1] = 20;
a[2] = 30;

free(a);
```

Alocação dinâmica de memória

- Memória estática

- Armazena instruções do programa, variáveis globais e estáticas
- É possível definir o tamanho da memória estática antes de executar o programa

- Memória dinâmica

- Armazena variáveis locais, pilhas de execução e memória alocada dinamicamente
- Tamanho da memória pode variar conforme fluxo de execução do programa
- A região de memória livre é utilizada para alocação dinâmica de memória
- Pode crescer ou diminuir de através das funções malloc, realloc ou free

memória estática	Código do programa
	Variáveis globais e Variáveis estáticas
memória dinâmica	Variáveis alocadas dinamicamente
	Memória livre
	Variáveis locais (Pilha de execução)

Estrutura

- Possibilita a criação de estruturas complexas
 - São formadas por um conjunto de atributos
- Instrução `struct`

```
struct <estrutura> {  
    <definição da estrutura>  
}
```

```
struct ponto2D {  
    int x;  
    int y;  
}
```

```
struct ponto p;  
p.x = 10;  
p.y = 5;
```

Definindo novos tipos de dados

- Definição de novos tipos baseados em estruturas
 - Permite a alocação dinâmica de uma maneira mais usual
 - Utiliza-se a instrução `typedef` para definição de novos tipos
 - O operador de união utilizado em alocação dinâmica é `->` ao invés de `.`
- Exemplo

```
typedef struct ponto2D {  
    int x;  
    int y;  
} Ponto2D;
```

```
Ponto2D *p = malloc(sizeof(Ponto2D));  
p->x = 10;  
p->y = 5;
```


Exercícios

1. Dado um vetor de números inteiros v de tamanho n e um número k , retorne verdadeiro se a soma de qualquer par de números em v for igual a k .
 - Exemplo: dado $v = [10, 15, 3, 7]$ e $k = 17$, a saída deve ser true, pois $10 + 7$ é 17
2. Dado um vetor de números inteiros v , retorne um novo vetor de forma que cada elemento no índice i seja o produto de todos os números na matriz original, com exceção de i .
 - Exemplo 1: dado $v = [1, 2, 3, 4, 5]$, a saída esperada é $[120, 60, 40, 30, 24]$
 - Exemplo 2: dado $v = [3, 2, 1]$, a saída esperada é $[2, 3, 6]$

Exercícios

3. Dado um vetor de números inteiros v , encontre o primeiro inteiro positivo ausente no vetor. Em outras palavras, deve ser retornado o menor inteiro positivo que não existe no vetor. A matriz pode conter duplicados e números negativos também. O algoritmo deve apresentar complexidade de tempo linear e de espaço constante (pode desconsiderar o esforço para ordenação do vetor).
 - Exemplo 1, dado $v = [3, 4, -1, 1]$, a saída esperada é 2
 - Exemplo 2, dado $v = [1, 2, 0]$, a saída esperada é 3

4. Dada um vetor inteiros v , retorne a maior soma dos números não adjacentes. Os números podem incluir 0 ou negativos no vetor.
 - Exemplo 1, dado $v = [2, 4, 6, 2, 5]$, a saída esperada é 13, considerando $2 + 6 + 5$
 - Exemplo 2, dado $v = [5, 1, 1, 5]$, a saída esperada é 10, considerando $5 + 5$

Exercícios

5. Considere uma escadaria com n degraus e você pode subir 1 ou 2 degraus por vez. Dado n , retorne o número de maneiras únicas de subir a escada.
- Exemplo, dado $n = 4$, existem 5 maneiras exclusivas
 - $[1, 1, 1, 1]$, $[2, 1, 1]$, $[1, 2, 1]$, $[1, 1, 2]$, $[2, 2]$
6. Dado um vetor de inteiros v com tamanho n e k com intervalo $1 \leq k \leq n$, calcule os valores máximos para cada subvetor de comprimento k gerado a partir do vetor v .
- Exemplo 1, dado $v = [10, 5, 2, 7, 8, 7]$ e $k = 3$, a saída será $[10, 7, 8, 8]$, visto que:
 - Exemplo 2, dado $v = [5, 1, 1, 5]$, a saída é 10, considerando $5 + 5$
 - $10 = \max(10, 5, 2)$
 - $7 = \max(5, 2, 7)$
 - $8 = \max(2, 7, 8)$
 - $8 = \max(7, 8, 7)$

Exercícios

7. Dadas duas listas encadeadas acíclicas de inteiros que se cruzam em algum ponto, localize o primeiro nó de interseção.
 - Exemplo, dado $A = 3 \rightarrow 7 \rightarrow 8 \rightarrow 10$ e $B = 99 \rightarrow 1 \rightarrow 8 \rightarrow 10$, a saída esperada será o valor 8
8. Dada uma série de colchetes, parênteses e chaves abertos ou fechados, retorne verdadeiro se a série estiver balanceada (bem formada).
 - Exemplo 1, dada a string "`([]) [] ({ })`", a saída deve ser `true`
 - Exemplo 2, dada a string "`([])`" ou "`((())`", a saída deve retornar `false`

Exercícios

9. *Run-length encoding* (RLE) é uma forma simples de compressão de textos. A ideia desta técnica é representar caracteres repetidos sucessivamente com um contador seguido pelo caractere. Dada uma *string*, retorne o texto resultante da aplicação da técnica RLE.
 - Exemplo, dada a string "AAAABBBCCDAA", a saída compactada deve ser "4A3B2C1D2A"
10. *Power set* é um conjunto gerado a partir da combinação de todos seus subconjuntos. Dado um conjunto v , retorne o *power set* deste conjunto de entrada.
 - Exemplo: dado $v = [1, 2, 3]$, a saída deve ser:
 - $[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]$

Revisão da linguagem C

Projeto de arquivos
Prof. Allan Rodrigo Leite