Treinando para a Maratona de Programação

André Amaral de Sousa
Athos Couto
Victor Colombo

30 de agosto de 2019

Introdução

Este projeto foi elaborado com o foco em ensinar o conteúdo básico necessário para competir na Maratona de Programação.

O Formato do curso consiste em aulas que cobrem todo o conteúdo básico, e segue uma ordem didática. Ao final de cada aula, haverá uma lista de exercícios com problemas de diversos juízes online, para um completo aprendizado é necessário resolver os problemas das listas.

Como a maioria dos competidores de alto nível do mundo inteiro utiliza C++ nas competições de programação, o curso será baseado nesta linguagem, e as primeiras aulas cobrirá todo o conteúdo necessário desta linguagem.

Este projeto foi patrocinado pela VTEX, e elaborado pelos ex-competidores André Amaral de Sousa, Athos Couto e Victor Colombo

Sumário

Capítulo 1

Linguagem C++

Conforme explicado na Introdução utilizaremos C++ como linguagem base para e portanto ensinaremos o conteúdo necessário para a Maratona de Programação.

1.1 Aula 1: Leitura e Impressão

Nesta aula iremos mostrar como ler e imprimir dados.

1.1.1 Primeiro programa

Para uma familiarização inicial com a linguagem, iremos analisar o seguinte programa:

```
1 #include <bits/stdc++.h>
2 int main() {
3    printf("Hello World!");
4    return 0;
5 }
```

Saída:
Hello World!

Análise das linhas importantes:

```
#include <bits/stdc++.h>
```

Essa linha basicamente inclui todas as bibliotecas padrão do C++, que contêm essencialmente todos os comandos que você utilizará em programação competitiva.

```
int main() {...}
```

Essa linha serve para criarmos a parte principal de nosso programa. Dentro das chaves é onde escrevemos o que queremos que o programa faça. O programa sempre começa a ser executado no primeiro comando dessa **main**.

Normamente adotamos padrões na forma de escrever nossos códigos para que fiquem legíveis, não há a menor diferença para o compilador (que transforma o nosso código em linguagem de máquina), mas é mais fácil encontrar os erros de códigos legíveis do que de códigos bagunçados. Portanto, note que o fecha chaves **}** está na mesma coluna que o **int main()**, e note também que os comandos dentro da **main** são facilmente identificados porque começam numa mesma coluna (para este espaço usamos a tecla tab).

Nos referimos a estes padrões através da palavra Indentação, que significa o espaço que se deixa entre a margem e o começo da linha escrita.

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

A função printf serve para escrever na tela (que é a saída padrão) alguma informação, que no caso do nosso programa é **Hello World!**.

Repare no ponto-e-vírgula ao final da linha. A maioria dos comandos em C++ tem de ser sucedidos por ';'(que indica o fim do comando).

```
return 0;
```

Este comando serve para indicar que o programa terminou sem erros.

1.1.2 Variáveis

As variáveis são espaços de memória onde armazenamos informações enquanto o programa estiver executando (ao fim do programa perdemos o controle da variável). Para um bom entendimento, utilizaremos a seguinte analogia, as variáveis são como caixas dispostas numa prateleira, onde podemos guardar coisas. Quando declaramos uma variável num programa, o computador procura nessa prateleira (que pela analogia seria a própria memória) uma caixa vazia que atenda ao nosso pedido, para isso precisamos dar um nome à variável e dizer qual o tipo que queremos, o nome serve para que você em seu programa identifique a caixa e o tipo se refere ao tipo de informação que você guardará na caixa (por exemplo um número inteiro, uma letra, etc) e também serve para o computador saber o tamanho da caixa. Vale lembrar que a prateleira é organizada, então a caixa tem um endereço na prateleira (para que o computador a identifique), este endereço é identificado através do caractere **&**, no momento ainda não usaremos ele, mas em breve será necessário.

Identificador:

Trata-se do nome que você escolhe para a variável. Conforme dito anteriormente, este nome é utilizado no programa para você identificar aquela variável. Existem algumas regras para os identificadores das variáveis:

- Usar somente letras(não usar acentos, nem cedilha), números e underlines (_)
- Iniciar com uma letra (por exemplo não se pode iniciar com um número)
- São permitidas letras maiúsculas e minúsculas (por exemplo A é diferente de a, ou seja, C++ é case sensitive)
- O comprimento não deve ser muito longo
- Não podem ser palavras reservadas como por exemplo main, int. As palavras reservada são:

  asm, auto, bool, break, case, catch, char, class, const, const_cast,continue, default, delete, do, double, dynamic_cast, else, enum,explicit, export, extern, false, float, for, friend, goto, if, inline,int, long, mutable, namespace, new, operator, private, protected,public, register, reinterpret_cast, return, short, signed, sizeof,static, static_cast, struct, switch, template, this, throw, true, try,typedef, typeid, typename, union, unsigned, using, virtual, void,volatile, wchar_t, while

  Observação: Essa lista só foi mostrada por curiosidade, e para eventual consulta, não se deve decorá-la. Outra observação importante é procurar dar nomes que tenham algum significado, que tenham algo a ver com o papel da variável.

Tipo:

Para que o computador encontre uma caixa adequada, na linguagem C++ precisamos informar o tipo de informações que essa variável guardará. Vale ressaltar que o computador trabalha apenas com zero e um (binário), e o espaço de memória que armazena zero ou um é o Bit (proveniente do termo Binary Digit), mas normalmente usamos o Byte (Binary Term) que é o espaço de memória equivalente a 8 bits, assim cada tipo ocupa um certo espaço na memória (na analogia seria o tamanho da caixa). Os principais tipos que usaremos são:

- **bool**: Armazena apenas verdadeiro: *true* ou falso: *false*. Normalmente possui o tamanho de 1 byte.
- **char**: Armazena caracteres, como por exemplo letras(maiúsculas e minúsculas), espaço, vírgula. Normalmente possui o tamanho de 1 byte. Vale observar que um caractere nada mais é que um número de 0 a 255, onde cada número está associado a um caractere através da tabela ASCII.
- **int**: Armazena um número inteiro, normalmente possui o tamanho de 4 bytes, assim pode armazenar os números inteiros de $-2^{31} = -2147483648$ a $(2^{31}) - 1 = 2147483647$.

- **unsigned int**: Armazena números inteiros sem o sinal, possui o mesmo tamanho do int, assim pode armazenar números inteiros de 0 a 4294967295.
- **long long int**: Armazena um número inteiro, normalmente possui o tamanho de 8 bytes, assim pode armazenar os números inteiros de $-2^{63} = -9223372036854775808$ a $(2^{63}) - 1 = 9223372036854775807$
- **float**: Armazena um número real, possui normalmente o tamanho de 4 bytes, assim possui precisão de aproximadamente 7 dígitos.
- **double**: Armazena um número real, possui normalmente o tamanho de 8 bytes, assim possui precisão de aproximadamente 15 dígitos.

Quando a informação não pode ser armazenada por ser maior que o tamanho da variável, dizemos que ocorreu Overflow, assim precisamos de uma caixa maior para esta informação, tome muito cuidado para não ocorrer overflow em seus programas (geralmente é um erro difícil de indentificar, e bem comum para iniciantes). E quando a informação não pode ser armazenada por ser muito pequena, dizemos que ocorreu Underflow.

Para um maior entendimento, analisemos o seguinte programa:

```
1  #include <bits/stdc++.h>
2
3  int main() {
4    //comentários são feitos assim
5    float myFloat = 1.92;
6    double myDouble;
7    int myInt;
8    char myChar = 'A';
9    myInt = 10;
10   myDouble = 1e-4;
11   printf("myFloat=%f myDouble=%f \nmyInt=%d
      ↪  myChar=%c\n",myFloat,myDouble,myInt,myChar);
12   return 0;
13 }
```

Saída:
myFloat=1.920000 myDouble=0.000100
myInt=10 myChar=A

Análise das linhas importantes:

```
//comentários são feitos assim
```

Comentários em C++ são feitos com `//` para uma linha ou `/* ... */` para múltiplas linhas.

```
float myFloat = 1.92;
```

Nesta linha estamos declarando uma variável do tipo **float** com nome (identificador) **myFloat**, e estamos armazenando o valor 1.92 nesta variável (obs.: devemos usar **.** para separar as cadas decimais e não **,** senão teremos um erro de compilação). Note que declarar uma variável e atribuir valor para ela são dois comandos, mas fizemos na mesma linha.

```
char myChar = 'A';
```

Nesta linha estamos declarando uma variável do tipo **char** com nome **myChar**, e estamos armazenando o caractere A nela, note que para que o compilador entenda que A é um caractere temos que colocar entre aspas simples.

```
myInt = 10;
```

Nesta linha estamos apenas atribuindo o valor 10 para a variável **myInt** que foi declarada anteriormente (na linha 7). É bom saber que logo após a declaração de uma variável, ela pode conter qualquer valor, pois é como se na declaração o computador encontrasse uma caixa que não está sendo usada por nenhum programa mas que tem algo dentro (sujeira, algo que não temos controle), por isso é importante sempre atribuir algum valor às nossas variáveis antes de usá-las.

```
myDouble = 1e-4;
```

Nesta linha o valor 0.0001 é armazenado na variável anteriormente declarada **myDouble**, note que 1e-4 é a notação científica de 0.0001 .

1.1.3 Comandos *printf* e *scanf*

Agora vamos aprender como pegar informações digitadas (através do teclado que é a entrada padrão) e como imprimir na tela (que é a saída padrão). Vimos no primeiro programa uma impressão simples, através do comando **printf**, estudaremos melhor as funções (por enquanto encare função como um comando, teremos uma aula para explicar o que são funções) **printf** e **scanf**.
Vale ressaltar que as funções **printf** e **scanf** são da linguagem C, mas que também funcionam em C++. As funções para leitura e impressão do C++ são **cin** e **cout**.

Função printf

A função **printf** possui a seguinte sintaxe:

```
printf("expressao de controle", argumentos);
```

Quando estudarmos funções conseguiremos entender melhor o que são argumentos, mas basicamente são as informações que passamos para a função. Na função **printf** os argumentos são as variáveis que queremos imprimir na tela (na verdade seu conteúdo), e com exceção de alguns caracteres especiais tudo que for escrito na expressão de controle sairá na tela.

Para aprender o funcionamento destes caracteres especiais, vamos analisar o seguinte programa:

```
1  #include <bits/stdc++.h>
2
3  int main() {
4     printf("*#TuDo IgUaL#*");
5     printf("pulou linha ??");
6     printf("\n e agora ?\n");
7     int a=10;
8     printf("temos a=%d dobro=%d\n",a,2*a);
9     printf("observe: \" \\");
10    return 0;
11 }
```

Saída:

```
*#TuDo IgUaL#*pulou linha ??
  e agora ?
temos a=10 dobro=20
observe: " \
```

Análise das linhas importantes:

```
printf("*#TuDo IgUaL#*");
```

Note que neste caso a expressão de controle foi impressa na tela exatamente como foi escrita.

```
printf("pulou linha ??");
```

Perceba que este printf imprimiu na mesma linha que o anterior.

```
printf("\n e agora ?\n");
```

Agora sim pulou linha, pois **n** é um caractere especial que pula linha.

```
printf("temos a=%d dobro=%d\n",a,2*a);
```

Observe que o primeiro **%d** foi substituído pelo conteúdo da variável a e o segundo foi substituído por **2*a**, isso ocorre porque a função **printf** substitui certos caracteres especiais pelos argumentos passados, e na ordem que foram passados. Nesse caso **%d** é um caractere especial para indicar que iremos substituir por um número inteiro, seja de uma variável do tipo **int** (como é o caso da variável **a**) ou de uma expressão (como é o caso do **2*a**). Outros caracteres especiais são:

- **%c**: Argumento do tipo **char**.
- **%lld**: Argumento do tipo **long long int**.
- **%f**: Argumento dos tipos **float** e **double**.

  Vale ressaltar que ao imprimir um número real (float ou double) com **%f** este sairá com seis dígitos decimais, usando **%.2f** este sairá com dois dígitos decimais, assim basta colocar entre **%.** e **f** a quantidade de dígitos que você deseja que seu número seja impresso, mas tome cuidado pois o último dígito pode ser arredondado se necessário (é recomendado que se façam testes para um melhor entendimento).

```
printf("observe: \" \\");
```

Este printf mostra que podemos imprimir certos caracteres com o auxílio do \, basicamente ele indica que o próximo caractere é pra ser usado sem seu significado padrão, e sim como caractere puro, note também que ele próprio não aparece na tela pois tem apenas esta função. Por exemplo se tentarmos imprimir “ sem \ ocorrerá um erro de compilação, pois o segundo “ fechará com o primeiro e a sintaxe do **printf** estará errada.

Função scanf

A função **scanf** nos permite ler dados da entrada padrão, e armazenar em variáveis do nosso código. Ela possui a seguinte sintaxe:

```
scanf("expressao de controle", argumentos);
```

Na função **scanf** os argumentos são os endereços das variáveis onde queremos armazenar as informações lidas (agora sim teremos que usar o **&** para identificar os endereços das variáveis).
Para entender o funcionamento da função **scanf**, vamos analisar o seguinte programa:

```
1 #include <bits/stdc++.h>
2
3 int main() {
4    int a;
5    scanf("%d",&a);
6    printf("a=%d",a);
7    return 0;
8 }
```

Entrada:
10

Saída:
a=10

Análise das linhas importantes:

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

Nesta linha estamos lendo um número inteiro, por isso colocamos **%d** na expressão de controle, já no argumento devemos colocar o local onde queremos armazenar este inteiro, no caso queremos armazenar na variável **a**, mas para isso precisamos passar como argumento o endereço da variável **a**, conforme explicado anteriormente basta usar **&a**. Um erro comum é esquecer de colocar **&**, fique atento !
Perceba que agora temos a parte da **Entrada**, ela representa o que foi digitado para o programa. Muitas IDEs (como por exemplo DevC++, Codeblocks, etc) ao executar um programa juntam na mesma tela a entrada e a saída do programa, isso pode ficar muito confuso para códigos grandes (um erro comum é apertar a tecla enter ao digitar a entrada, o que pula linha, e achar que a saída está pulando linha, quando na verdade não está), o recomendável é separar a entrada da saída, como faz o editor online CS Academy (note as abas input e output), mas também é possível executar seu código através do terminal,

criar um arquivo de texto contendo sua entrada e direcionar este arquivo para a entrada do seu programa.
Outra particularidade da entrada é que ela é consumida pelo programa, no seguinte sentido, cada caractere da entrada pode ser lido ou descartado, e depois nunca mais volta a ser considerado (não conseguimos ler um conteúdo da entrada novamente). Por exemplo, se na entrada tivéssemos ␣**10**, este espaço seria descartado pois estamos lendo um inteiro e espaço não faz parte da representação de um número inteiro. Por isso é extramamente importante entender o formato da entrada dos exercícios e seguí-los à risca (e da saída também, já que a correção é automatizada).
Também podemos ler mais de uma informação no mesmo comando **scanf**, basta colocar vírgula entre os argumentos, por exemplo `scanf("\%d \%d",&a, &b);` E para ler outros tipos de variáveis devemos usar os seguintes caracteres especiais:

- **%c**: Argumento do tipo **char**.
- **%lld**: Argumento do tipo **long long int**.
- **%f**: Argumento do tipo **float**.
- **%lf**: Argumento do tipo **double** (tome cuidado é diferente do **printf**).

Uma particularidade da função **scanf** é que ela retorna o número de variáveis lidas (assim como em matemática uma função recebe argumentos e retorna algo, as vezes pode não retornar nada, mas no caso do **scanf** retorna um inteiro, o número de variáveis lidas, na parte de funções isto ficará mais claro). Por exemplo se tivermos `printf("\%d",scanf("\%f \%lf",&f,&d));` e digitarmos dois números reais (um **float** e um **double**), este **printf** imprimirá 2, pois o **scanf** lerá duas variáveis.

Lista de Exercícios:

- JPNEU
- QUADRAD2
- PEDAGIO1

1.2 Aula 2: Estrutura condicional (comando *if/else*)

Nesta aula iremos mostrar o primeiro comando que controla o fluxo do código. A princípio a execução do código começa pelo **int main(){...}** e sempre vai para a próxima linha, através da estrutura condicional (utilizando o comando if/else) isso nem sempre será verdade.
Antes de aprender o comando **if/else** precisamos aprender a escrever condições (expressões booleanas), por isso iremos estudar os operadores disponíveis em C++.

1.2.1 Operadores Aritméticos e Operadores de Atribuição

Primeiro iremos estudar como escrever expressões aritméticas.

Operadores Aritméticos

Na linguagem C++ temos os seguintes operadores aritméticos:

- +: operador para adição.
- **-**: operador para subtração.
- ***: operador para multiplicação.
- /: operador para divisão.
- %: operador para o resto da divisão inteira (ex.: o resto da divisão de 14 por 6 é 14%6 = 2).

A ordem de prioridade dos operadores comentados acima é a seguinte:

Maior prioridade: *, /, %.

Menor prioridade: +, **-**.

Para operadores com mesma ordem de prioridade as operações são realizadas da esquerda para a direita, por exemplo a expressão $1+2*5\%3$ é realizada da seguinte forma, os operadores com maior precedência são * e %, mas * aparece primeiro (considerando o sentido da esquerda para a direita) então a primeira operação a ser feita é **2*5=10**, com isso ficamos com **1+10%3**, e como textbf% tem maior prioridade que textbf+, então a segunda operação a ser feita é textbf10%3=1, e por último ficamos com **1+1=2**, logo **1+2*5%3=2**. Mas podemos mudar a ordem das operações se usarmos parênteses (é recomendado sempre usar parêntheses para garantir que a expressão é a desejada), por exemplo na expressão **(1+2)*(5%3)** a primeira operação a ser feita é **1+2=3** pois está entre parênteses e aparece mais à esquerda que **(5%3)**, assim temos **3*(5%3)**, e portanto a segunda operação a ser feita é **5%3=2** pois está entre parênteses, logo teremos **3*2=6** e portanto **(1+2)*(5%3)=6**.

Operadores de Atribuição

Na linguagem C++ quando queremos atribuir um valor a uma variável, ou seja, alterar o conteúdo desta variável usamos o comando de atribuição =. Por exemplo o seguinte comando **a = 5**; significa que, devemos jogar fora o conteúdo de a (limpar a variável) e depois colocar o valor 5 nela, e então variável a passa a valer 5. Outros operadores de atribuição nos possibilitam reduzir atribuições da seguinte forma:

Forma Normal `variavel = variavel operador expressao;`	**Forma Reduzida** `variavel operador= expressao;`
`a = a + b;`	`a += b;`
`a = a - b;`	`a -= b;`
`a = a * b;`	`a *= b;`
`a = a / b;`	`a /= b;`
`a = a % b;`	`a %= b;`

Observe que num comando do tipo **a = a+b;** onde inicialmente **a** vale **5** e **b** vale **2**, é realizado da seguinte maneira, primeiro calculamos a expressão que devemos atribuir à variável, no caso **a+b**, mas **a** por enquanto vale **5** por isso **a+b** dá **7** e só então limpamos a variável **a** e colocamos o valor **7** nela.
Outros operadores úteis são os operadores de incremento, que soma **1** na variável, e decremento, que subtrai **1** na variável:

	Operador de Incremento ++	Operador de decremento --
pós-fixado:	`a++;`	`a--`
pré-fixado:	`++a;`	`--a`
equivalente:	`a = a + 1;`	`a = a - 1;`

A diferença entre a forma pós-fixada e a forma pré-fixada está na ordem que ocorre o incremento (ou decremento) em relação a outros comandos, por exemplo **b = ++a;** é equivalente aos comandos: **a = a+1;** depois **b = a;** (nessa ordem), enquanto que **b = a++;** é equivalente aos comandos **b = a;** depois **a = a+1;** (nessa ordem).

Para uma maior compreensão vamos analisar o seguinte código:

```
1  #include <bits/stdc++.h>
2
3  int main() {
4     int a,b; scanf("%d %d",&a,&b);
5     printf("entrou a = %d b = %d\n",a,b);
6     int c = a + b;
7     printf("a = %d e b=%d nao mudaram!!\n",a,b);
8
9     c = a/b;
10    double d = a/b;
11    printf("c=%d d=%f\n",c,d);
12    d = (double) a/b;
13    printf("d=%.3f\n",d);
14
15    c -= 10;
16    c %= 3;
17    printf("c = %d\n", c);
18
19    return 0;
20 }
```

Entrada:
5 2

Saída:
entrou a = 5 b = 2
a = 5 e b=2 nao mudaram!!
c=2 d=2.000000
d=2.500
c = -2

Análise das linhas importantes:

```
int c = a + b;
```

Nesta linha estamos atribuindo à variável **c** o valor de **a+b**, na analogia onde as variáveis são caixas, podemos entender que somamos os conteúdos das caixas **a** e **b** sem alterá-las (o valor de **a** não muda e o de **b** também não), jogamos fora o que estava na caixa **c** (limpando a caixa) e então colocamos tal soma na caixa **c**.

```
c = a/b;

double d = a/b;

printf("c=%d d=%f\n",c,d);
```

Quando dividimos **a** por **b**, estamos dividindo um número do tipo **int** por outro do tipo **int**, e uma característica dos operadores mostrados acima é que o tipo da resposta depende dos tipos envolvidos, neste caso a resposta necessariamente será do tipo **int**, é como se dividissemos o valor de **a** que é **5** pelo de **b** que é **2** resultando em **2.5** mas pela propriedade deste operador temos que retornar um inteiro, e como não conseguimos armazenar **2.5** num inteiro pegamos apenas a parte inteira (matematicamente estamos pegando $\lfloor a/b \rfloor$, o piso da divisão de **a** por **b**) que neste caso é **2**. Sempre que tivermos dois tipos envolvidos numa operação a resposta sairá naquele que tem mais capacidade de armazenamento (aquele que é 'maior', por exemplo **long long int** é 'maior' que **int**). Tendo isto em mente, vemos que não adianta querermos colocar o

resultado da divisão de **a** por **b** numa variável do tipo **double**, no exemplo a variável **d**, ainda assim o resultado será **2** e não **2.5**, pois após dividir **a** por **b** que retorna **2** e aí se colocamos o inteiro **2** numa variável tipo **double** vira **2.000000**.

```
d = (double) a/b;

printf("d=%.3f\n",d);
```

Tendo em vista o que foi apresentado acima, podemos calcular corretamente o valor de **a/b** se encararmos uma dessas variáveis como **double**, é como se pegássemos uma caixa maior que a do **int** (nesse caso a do **double**) copiasse o conteúdo de **a** sem alterar a variável **a** e agora sim dividir por **b**, e como **double** é 'maior' que **int**, a resposta da divisão sairá em **double**, e portanto teremos **2.500000** como queríamos. Outra forma equivalente seria **d=a/(double) b;** assim encararíamos **b** como **double**. E perceba que **d=(double)(a/b);** está errado, ou seja , dará **2.000000** pois estamos primeiro fazendo a divisão para depois encarar como **double** o que é equivalente ao que fizemos antes **d=a/b;**. Este processo de encarar uma variável de um tipo como de outro tipo é o que chamamos de **Type Casting** (também chamado apenas de cast), e para isso ocorrer basta colocar entre parênteses o tipo que deseja encarar logo antes da variável (conforme no exemplo).

```
c -= 10;

c %= 3;

printf("c = %d\n", c);
```

Note que o valor da variável **c** antes destas linhas é **2**, após o comando **c -= 10;** o valor de **c** passa a ser **-8**, e então temos o comando **c %= 3;** que deve calcular o resto da divisão de **-8** por **3** e colocar na variável **c**, acontece que o operador **%** mantém o sinal, ou seja, como **-8** é negativo, o resto calculado também será negativo, e portanto o resultado é **-2**, tome muito cuidado com isso. Apesar destes detalhes, algo que sempre vale é: **a = (a/b)*b + a%b**.

1.2.2 Operadores Relacionais e Operadores Lógicos

Agora iremos estudar como escrever expressões booleanas.

Operadores Relacionais

Para fazer uma comparação entre duas expressões utilizamos os seguintes operadores relacionais:

- **==**: Igual.
- **!=**: Diferente.
- **>**: Maior.
- **>=**: Maior ou Igual.
- **<**: Menor.
- **<=**: Menor ou Igual.

Quando usamos um operador relacional o computador encara como uma afirmação, uma variável do tipo **bool**, assim se ela estiver verdadeira receberá valor **true** e se estiver falsa receberá valor **false**. Mas também é possível encarar uma afirmação como um inteiro, esse cast é automático (não é necessário colocar **(int)** para usar uma expressão como um inteiro, esse é um dos motivos pelos quais dizemos que C++ não é fortemente tipada), assim se ela estiver verdadeira receberá valor **1** e se for falsa receberá **0**.

Operadores Lógicos

Quando a expressão booleana que queremos escrever depende de mais de uma afirmação, podemos combinar tais afirmações com operadores lógicos de forma a gerar condições mais complexas, os operadores lógicos são:

- **!**: Operador Negação.
  É utilizado da seguinte forma: **!condição**, ele inverte o valor da **condição**, por exemplo se **condição** for verdadeira (valor 1) retorna falso (valor 0) enquanto que se **condição** for falsa (valor 0) retorna verdadeiro (valor 1).

- **&&**: Operador E.
  É utilizado da seguinte maneira: **condição1 && condição2**, e retorna verdadeiro (valor 1) apenas se a **condição1** E a **condição2** forem verdadeias, caso contrário retorna falso (valor 0).

- ||: Operador OU.
  É utilizado da seguinte maneira: **condição1** || **condição2**, e retorna falso (valor 0) apenas se a **condição1** e a **condição2** forem falsas, caso contrário retorna verdadeiro (valor 1), ou seja, retorna verdadeiro se a **condição1** OU a **condição2** for verdadeira (OU inclusivo, assim se **condicão1** for verdadeira e a **condição2** for verdadeira também, retorna verdadeiro, valor 1).

Para uma maior compreensão vamos analisar o seguinte código:

```cpp
#include <bits/stdc++.h>

int main() {
   int a,b; scanf("%d %d",&a,&b);
   printf("%d\n", 2*5 - (3 <= 6));
   printf("%d\n", 20/3 == 6);
   printf("%d\n", 5/2 == 2.5);
   printf("%d\n", 5<=7 && 12*3 == 36);
   printf("%d %d\n", !(a%2), b != 2*a);
   printf("%d\n",!(a%2) || (b != 2*a));
   return 0;
}
```

Entrada:
4 5

Saída:
9
1
0
1
1 1
1

Análise das linhas importantes:

```cpp
printf("%d\n", 2*5 - (3 <= 6));
```

Como $3 <= 6$ é verdadeiro, temos que seu valor inteiro é **1** (note que como estamos imprimindo com **%d** então temos que encarar como inteiro), logo temos que $2*5-(3 <= 6) = 2*5-1 = 9$. Observe que o valor de $(3 <= 6)$ foi calculado primeiro pois está entre parênteses, mas a prioridade dos operadores que vimos é a seguinte:

Prioridade dos Operadores
()
++, –, **!**, **(type)**
%, *, /
+, -
>, >=, <, <=
==, !=
&&
||
=, **operador=**

Entre operadores com mesma prioridade o cálculo é realizado da esquerda para a direita, com exceção dos seguintes operadores: ++, –, **!**, **(type)**, = e **operador=** , que são processados da direita para a esquerda. Vale ressaltar que isso não deve ser decorado, use esta tabela como referência, e sempre que estiver em dúvida coloque parêntheses (que possui a maior prioridade) e teste.

```cpp
printf("%d\n", 20/3 == 6);
```

```
printf("%d\n", 5/2 == 2.5);
```

Lembre-se que operações entre inteiros por padrão resulta em um inteiro, portanto o resultado de $20/3$ será armazenado em um inteiro e assim dará **6**, portanto a expressão $20/3 == 6$ será verdadeira. De forma análoga, $5/2$ dará **2** e assim $5/2 == 2.5$ será falso.

```
printf("%d\n", 5<=7 && 12*3 == 36);
```

Nesta linha estamos usando o operador , que como vimos retorna verdadeiro apenas se as duas expressões forem verdadeiras, e como $5 <= 7$ e $12 * 3 == 36$ são ambas verdadeiras então a expressão $5 <= 712 * 3 == 36$ é verdadeira. vale ressaltar que, caso a primeira condição de um operador seja falsa, então a expressão será falsa e a segunda condição nem será calculada, por exemplo $7 <= 52/0 == 36$ como $7 <= 5$ é falso, então a segunda condição do operador nem é calculada, pois já é possível determinar que a expressão inteira será falsa. De forma análoga se a primeira condição do operador —— for verdadeira, então a segunda condição nem é calculada e a expressão como um todo será verdadeira.

```
printf("%d %d\n", !(a%2), b != 2*a);
```

Nesta linha podemos ver que um número também pode ser encarado como uma afirmação, onde será falso se seu valor for **0** e será verdadeiro caso contrário. Assim como no exemplo o valor da variável **a** é **4**, então $a\%2$ resulta no valor **0** que é considerado falso, mas usando o operador **!** passa a ser verdadeiro, e como estamos imprimindo como inteiro, então verdadeiro é impresso como **1**.

1.2.3 Comando *if/else*

Até o momento em todos os programas apresentados todas as linhas eram executadas em ordem, ou seja, após executar uma determinada linha passávamos para a execução da linha logo abaixo. A partir de agora iremos ver como controlar melhor o fluxo do código.

Comando *if*

O comando *if* possui a seguinte sintaxe:

```
if(condição) {
    comandos;
}
```

Seu funcionamento é, caso a **condição** seja verdadeira então os comandos dentro do bloco do **if** serão executados; E caso seja falsa nenhum comando dentro do bloco do **if** será executado, e a execução passa para a próxima linha depois do bloco do **if**.

Comando *if/else*

O comando *if/else* possui a seguinte sintaxe:

```
if(condição) {
    comandos1;
} else {
    comandos2;
}
```

Seu funcionamento é, caso a **condição** seja verdadeira então os comandos dentro do bloco do **if** serão executados (**comandos1**), e ao final da execução destes comandos a execução passa para a próxima linha depois do comando **if/else** inteiro, ou seja, pulamos o bloco do **else**; E caso a **condição** seja falsa então pulamos o bloco do **if** e os comandos dentro do bloco do **else** serão executados (**comandos2**), e ao final a execução passa para a próxima linha depois do comando **if/else** inteiro.

Perceba que o comando **if** é apenas o comando **if/else** sem a parte do **else** pois ela é opcional.

Múltiplos *if/else*

Muitas vezes dentro do **else** queremos colocar outro **if/else**, para isso podemos usar a seguinte sintaxe:

```
if(condição1) {
    comandos1;
} else if(condição2) {
    comandos2;
} else if(condição3) {
    comandos3;
    ...
} else {
    comandosN;
}
```

O funcionamento destes múltiplos comandos **if/else** é o seguinte, se a **condição1** for verdadeira apenas **comandos1** serão executados e depois a execução passa para a próxima linha depois do comando inteiro (logo após o bloco do **else**). Caso a **condição1** seja falsa e a **condição2** seja verdadeira, de forma semelhante, apenas **comandos2** serão executados. Assim por diante, e se nenhuma **condição** for verdadeira, então o bloco do **else** (**comandosN**) será executado, note que este **else** também é opcional.

Recomendamos que todos os comandos dentro um bloco (representado pelas chaves: **{** e **}**) estejam alinhados dentro do bloco, isso é o que chamamos de Identação, e ajuda a identificar melhor o que está dentro do bloco, para o compilador não faz diferença mas para humanos um código identado é muito melhor que entender. Vale ressaltar que para identarmos um código normalmente usamos a tecla tab e não o espaço.

Para uma maior compreensão vamos analisar o seguinte código:

```
1  #include <bits/stdc++.h>
2
3  int main() {
4    int a,b,c; scanf("%d %d %d",&a,&b,&c);
5
6    if(!(a>b && b>c)) {
7      a += b;
8      b += c;
9      printf("a=%d b=%d c=%d\n",a,b,c);
10   }
11
12   if(a%5 == 0)  printf("forma reduzida\n"),
     ↪  printf("a eh multiplo de 5\n");
13
14   if(b>10) {
15     a=b;
16   } else if(a>c) {
17     a=c;
18     printf("Conferindo se entrou\n");
19   }
20   return 0;
21 }
```

Entrada:
2 3 4

Saída:
a=5 b=7 c=4
forma reduzida
a eh multiplo de 5
Conferindo se entrou

Análise das linhas importantes:

```
if(a%5 == 0)  printf("forma reduzida\n"), printf("a eh multiplo de 5\n");
```

Além das sintaxes apresentadas, o comando **if/else** possui ainda sua forma reduzida, onde não usamos as chaves para determinar um bloco, apenas colocamos os comandos em uma única linha separados por vírgula. Esta forma reduzida é recomandada apenas quando o número de comandos é pequeno e cabem em uma linha, senão o código pode ficar muito confuso.

```
if(b>10) {
    a=b;
} else if(a>c) {
    a=c;
    printf("Conferindo se entrou\n");
}
```

Aqui quero mostrar uma das principais estratégias para encontrar erros num código, processo que chamamos de debugar. Consiste em colocar **printf**s para conferir a execução do código, com isso você sabe exatamente qual o fluxo do código e consegue encontrar o erro. Normalmente escrever códigos grandes sem nenhum erro é muito difícil, por isso devemos aprender a debugar rápido, o que exige muita prática, perceba também que um código bem identado facilita o processo de debugar, e portanto você deve se acostumar a identar bem seus códigos.

Lista de Exercícios:

- Flíper
- OVERF09
- NOTA09
- CONTA1
- MARCIAN1
- TRIANG11
- CAPITA13

1.3 Aula 3: Estrutura de repetição (comandos *while* e *for*)

Através do comando *if/else* conseguimos controlar o fluxo do nosso código de forma a executar ou não uma sequência de comandos, outra forma de controlar o fluxo do código é fazê-lo repetir uma sequência de comandos (também chamado de **loop**), o que é feito através das estruturas de repetição.

Comando *while*

O comando *while* possui a seguinte sintaxe:

```
while(condição) {
    comandos;
}
```

Seu funcionamento é, caso a **condição** seja verdadeira então os comandos dentro do bloco do **while** serão executados e ao final a **condição** é verificada novamente, ou seja, enquanto a **condição** for verdadeira os comandos dentro do bloco do **while** serão executados, e quando a **condição** se tornar falsa então a execução passa para a próxima linha depois do bloco do **while**. Chamamos de iteração cada vez que a sequência de comandos de um loop é executada.

Comando *for*

O comando *for* possui a seguinte sintaxe:

```
for(inicialização; condição; incremento) {
    comandos;
}
```

Seu funcionamento é, primeiramente são executados os comandos que estão na parte da **inicialização**, e estes comandos nunca mais serão executados novamente. Depois a **condição** é verificada e se for verdadeira os comandos dentro do bloco do **for** são executados e ao final os comandos da parte de **incremento** são executados, e então a **condição** é verificada novamente e entramos num loop. Quando a **condição** se tornar falsa então a execução passa para a próxima linha depois do bloco do **for**.

Apesar de termos estes dois comandos para criar uma estrutura de repetição, existe uma equivalência entre eles, ou seja, o comando **for** pode ser escrito como um comando **while**, como mostrado a seguir:

```
inicialização;
while(condição) {
    comandos;
    incremento;
}
```

Perceba que este comando **while** possui o mesmo funcionamento que um comando **for**, mas o comando **for** é mais compacto, pois em uma única linha conseguimos entender a **inicialização**, **condição** e **incremento**. Mas já que estes comandos são equivalentes, quando devemos usar cada um deles ? Não há uma regra, mas nossa recomendação é utilizar o comando **for** quando souber exatamente o número de iterções desejadas, e utilizar o comnando **while** quando não souber, ficará mais claro nos exemplos.

Comandos auxiliares *break* e continue

Existem dois comandos auxiliares para ajudar a controlar a execução dos loops, são eles os comandos **break** e **continue**. Eles podem ser usados em qualquer loop (seja com o comando **while** ou com o **for**).

O comando **break** funciona da sequinte forma:

```
while(condição1) {
    comandos1;
    if(condição2) break;
    comandos2;
}
```

Ao executar o comando **break** o loop atual é interompido, e apenas o loop atual, ou seja, se existirem vários loops encadeados apenas o atual será interrompido, equanto que os outros continuarão. Assim no exemplo acima, caso estejamos numa determinada iteração do **while** e a **condição2** seja verdadeira, então o **break** será executado, o que irá interromper o **while**, e assim a execução passará para a próxima linha depois do bloco do **while**, note que nesta iteração **comandos1** serão executados mas **comandos2** não serão executados.

Já o comando **continue** funciona da sequinte forma:

```
while(condição1) {
    comandos1;
    if(condição2) continue;
    comandos2;
}
```

Ao executar o comando **break** o loop atual pula para a próxima iteração, isso ocorre apenas no loop atual, ou seja, se existirem vários loops encadeados apenas o atual irá para a próxima iteração, equanto que os outros continuarão na iteração atual. Assim no exemplo acima, caso estejamos numa determinada iteração do **while** e a **condição2** seja verdadeira, então o **continue** será executado, o que irá pular **comandos2** e irá retornar para a verificação da **condição1**.

Para uma maior compreensão vamos analisar o seguinte código:

```
#include <bits/stdc++.h>

int main() {
  int num = 1, it = 0;
  while(num != 10) it++, num++, num *= 2;
  printf("iteracoes = %d\n",it);

  int soma = 0;
  for (int i = 1; i <= 10; i++) {
    soma += i;
  }
  printf("soma = %d\n",soma);

  bool ok = true;
  for(int i = 1; i <= 100 && ok == true; i++)
    for(int j = i + 1; j <= 100 && ok == true;
     ↪  j++)
      if(((j % i) == 0) && (j + i == 117)) {
        printf("i = %d j = %d\n", i, j);
        ok = false;
      }


  for(int a = 1; a <= 100; a++)
    for(int b = a + 1; b <= 100; b++)
      for(int c = b + 1; c <= 100; c++) {
        if(c % a != 0) continue;
        if(c % b != 0) continue;
        if(b % a != 0) continue;
        if(a + b + c <= 150) continue;
        printf("a = %d b = %d c = %d\n", a, b, c);
      }

  return 0;
}
```

Saída:

iteracoes = 2
soma = 55
i = 39 j = 78
a = 1 b = 50 c = 100
a = 2 b = 50 c = 100
a = 5 b = 50 c = 100
a = 7 b = 49 c = 98
a = 8 b = 48 c = 96
a = 10 b = 50 c = 100
a = 12 b = 48 c = 96
a = 16 b = 48 c = 96
a = 22 b = 44 c = 88
a = 23 b = 46 c = 92
a = 24 b = 48 c = 96
a = 25 b = 50 c = 100

Análise das linhas importantes:

```
int num = 1, it = 0;
while(num != 10) it++, num++, num *= 2;
printf("iteracoes = %d\n",it);
```

Assim como o comando **if/else**, os comandos **while** e **for** também possuem a forma reduzida onde se omite as chaves e os comandos são separados por vírgula. Perceba que não sabemos quantas vezes iremos executar a sequência de comandos dentro do loop, e por isso utilizamos um comando **while** (seguindo a recomendação dada). Perceba que se trocarmos o valor inicial da variável **num** para **2** ao invés de **1**, então o valor de **num** nunca chegaria a ser **10**, o que significa que nunca sairíamos do comando **while**, isto é o que chamamos de **loop infinito**. Tome muito cuidado com **loop infinito**, pois muitas vezes é difícil de identificar que seu código pode cair num **loop infinito**.

```
int soma = 0;
for (int i = 1; i <= 10; i++) {
    soma += i;
}
printf("soma = %d\n",soma);
```

Neste trecho queremos encontrar a soma dos números inteiros de 1 à 10, por isso optamos por utilizar o comando **for** já que sabemos exatamente quantas vezes queremos passar pelos comandos do loop. Observe que a variável **i** é uma auxiliar que nos informa em qual iteração estamos, por exemplo, quando o valor de **i** for **5** sabemos que estamos na quinta iteração deste loop, é muito comum utilizar uma variável auxiliar desta forma, procure entender a fundo este trecho pois ele será muito útil. Vale ressaltar que a variável **i** foi declarada dentro do bloco do comando **for**, e portanto esta variável só existe dentro do bloco deste **for**, o que significa que não há problema em ter outro comando **for** utilizando outra variável de mesmo nome **i** (desde que não seja dentro do bloco deste comando **for**), e de fato utilizaremos nas seguintes linhas.

```
bool ok = true;
for(int i = 1; i <= 100 && ok == true; i++)
    for(int j = i + 1; j <= 100 && ok == true; j++)
        if(((j % i) == 0) && (j + i == 117)) {
            printf("i = %d j = %d\n", i, j);
            ok = false;
        }
```

Este trecho de código é um pouco mais complicado, vou explicar com detalhes. Primeiro o intuito deste trecho é encontrar apenas um par de números distintos onde cada um deles seja no mínimo 1 e no máximo 100, o maior seja múltiplo do menor, e a soma deles dê 117. Para isso vamos fixar que o menor destes números será representado pela variável **i** e o maior por **j**. Como o valor de **i** está entre 1 e 100 então temos um loop que testará cada um destes valores para **i**, portanto fizemos um comando **for** que inicializa **i** com 1 e vai até o valor 100. Por enquanto ignore a variável **ok**, explicarei logo mais. Para cada valor de **i** quero testar todos os valores de **j**, e como $j > i$ então o menor valor possível de **j** é $\mathbf{i + 1}$, e o maior é 100, portanto fizemos um comando **for** que inicializa **j** com $\mathbf{i + 1}$ e vai até o valor 100. Note que estes dois comandos **for** estão em sua forma reduzida, faz sentido pois há apenas um comando dentro de cada um deles. Agora que estamos passando por todas as possibilidades do par **i, j** temos que checar as condições, e este é o papel do comando **if**, ele verifica se o resto de **j** dividido por **i** é igual a 0 (o que significa verificar se **j** é múltiplo de **i**) e verifica se a soma deles dá 117, caso estas duas condições sejam verdadeiras (note o uso do operador **&&**) então imprimimos os valores de **i** e **j**. Lembre-se que queremos apenas um par de números que satisfaça as condições dadas, então na primeira vez que estivermos dentro deste comando **if** queremos sair dos dois loops, note que não podemos usar o comando **break** pois ele sai apenas do loop atual (ou melhor do mais interno, no caso o **for** do **j**), e é por isso que usamos a variável auxiliar **ok**. Este é um truque que podemos usar sempre que precisarmos sair de vários loops encadeados, basicamente utilizamos uma variável auxiliar para guardar se ainda devemos continuar nos loops e acrescentamos ela na condição dos loops, no caso **ok** começa com **true** e quando determinamos o primeiro par de números trocamos **ok** para **false**, e assim os loops irão parar pois a condição deles não é mais válida.

```
for(int a = 1; a <= 100; a++)
    for(int b = a + 1; b <= 100; b++)
        for(int c = b + 1; c <= 100; c++) {
            if(c % a != 0) continue;
```

```
            if(c % b != 0) continue;
            if(b % a != 0) continue;
            if(a + b + c <= 150) continue;
            printf("a = %d b = %d c = %d\n", a, b, c);
    }
```

Se você conseguiu entender bem o trecho de código anterior, este aqui será mais fácil. O intuito deste trecho é gerar todas as triplas ordenadas de números distintos tais que, para todo par entre eles o maior seja múltiplo do menor, a soma dos três seja maior que 150, e cada um deles esteja entre 1 e 100. Podemos chamar o menor de **a**, o elemento médio de **b** e o maior de **c**. Assim levando em conta apenas que $a \leq b \leq c$ e que cada um deles deve ser no mínimo 1 e no máximo 100, então fizemos 3 comandos **for**s encadeados. Para garantir as outras condições, ao invés de fazer um comando **if** com as condições que queremos, como são muitas e ficaríam ruim em uma única linha, optamos por desconsiderar a tripla que não satisfaz alguma das condições, ou seja verificamos se a negação de alguma das condições é verdadeira, e se for utilizamos o comando **continue** para descartar esta tripla e ir para a próxima (este é um grande uso do comando **continue**), e assim apenas se todas as condições forem verdadeiras é que imprimiremos a tripla. Note que neste caso queremos listar todas as triplas ordenadas, por isso que não utilizamos o truque de sair dos loops encadeados ao encontrar a primeira tripla.

Lista de Exercícios:

- FATORIA2
- SOMA
- OBI8
- Número de Envelopes
- JDESAF12
- COFRE
- JESCADA

1.4 Aula 4: Vetores

Chamamos de Vetor (ou também array) um conjunto de variáveis que se localizam num espaço contínuo da memória, retomando a analogia de variáveis como caixas dispostas numa prateleira, um vetor é uma sequência de caixas uma ao lado da outra, onde identificamos a sequência toda com um nome e cada variável é identificada através de um índice.

Declaração

Para declarar um vetor usamos a seguinte sintaxe:

```
tipo nome[tamanho];
```

Note que todas as variáveis de um vetor possuem o mesmo tipo, e vale ressaltar que os índices válidos de um vetor são de **0** à **tamanho - 1**. Por exemplo:

```
int v[5];
```

Aqui estamos declarando um vetor de inteiros, com tamanho 5, cujo nome é v, então estamos declarando as seguintes variáveis: v[0], v[1], v[2], v[3], v[4]. Vale destacar que cada uma destas variáveis se comporta como uma variável int qualquer, ou seja, tudo que fazemos com uma variável int podemos fazer com qualquer uma delas. Tome muito cuidado para não acessar índice inválido, por exemplo v[-1] ou v[5], você só pode acessar as posições que você declarou (no caso os índices de 0 à 4), senão seu código poderá gerar um erro em tempo de execução (o que em juízes online é identificado como **Runtime Error** ou simplesmente **RE**). A grande vantagem de usar vetores é lidar com muitas variáveis (lidando com vários dados) de forma simples, ficará mais claro nos exemplos de código.

Vetores Multidimensionais

Podemos declarar vetores com mais de uma dimensão usando a seguinte sintaxe:

```
tipo nome[tamanho dimensão 1][tamanho dimensão 2]...[tamanho dimensão d];
```

Apesar de termos várias dimensões, na memória todas as variáveis de um vetor multidimensionais são seguidas. Note que cada índice deve estar entre **0** e **tamanho da dimensão - 1**, portanto estamos declarando um total de **(tamanho dimensão 1) * (tamanho dimensão 2)* ... * (tamanho dimensão d)** variáveis, e assim devemos ter cuidado para não exceder o limite de memória disponível ao nosso programa. Por exemplo:

```
int m[2][3];
```

Aqui estamos declarando um vetor de 2 dimensões, o que chamamos de matriz pois podemos usar como se estivessem dispostas da seguinte forma numa matriz:

m[0][0]	m[0][1]	m[0][2]
m[1][0]	m[1][1]	m[1][2]

Mas na realidade na memória estará em sequência da seguinte forma:

m[0][0]	m[0][1]	m[0][2]	m[1][0]	m[1][1]	m[1][2]

Para uma maior compreensão vamos analisar o seguinte código:

```cpp
#include <bits/stdc++.h>
const int MAXLOCAL = 100000;
const int MAXGLOBAL = 100000000;
const int MAXM = 1010;
const int MAXVAL = 15 + 1;
int vetorGlobal[MAXGLOBAL];
int matriz[MAXM][MAXM];
int marc[MAXVAL];
int main() {
  int n; scanf("%d", &n);
  int v[n];
  for(int i = 0; i < n; i++) scanf("%d", &v[i]);

  int vetorLocal[MAXLOCAL];
  printf(" MAXLOCAL = %d\n",MAXLOCAL);
  printf("MAXGLOBAL = %d\n",MAXGLOBAL);

  int m; scanf("%d",&m);
  for(int i = 0; i < m; i++)
    for(int j = 0; j < m; j++)
      scanf("%d", &matriz[i][j]);

  printf("Matriz Transposta:\n");
  for(int j = 0; j < m; j++) {
    for(int i = 0; i < m; i++)
      printf("%d ",matriz[i][j]);
    printf("\n");
  }

  int meuVetor[] = {-2,1,7,6};
  for(int i = 0; i < 4; i++)
    printf("meuVetor[%d] = %d\n",i,meuVetor[i]);

  int tam; scanf("%d", &tam);
  for(int i = 0; i < tam; i++) {
    int num; scanf("%d", &num);
    marc[num]++;
  }
  for(int val = 1; val < MAXVAL; val++)
    printf("A quantidade de %d eh %d\n", val,
    ↪  marc[val]);

  return 0;
}
```

Entrada:
3
5 6 7
4
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
8
1
3
5
5
3
3
2
15
Saída:
MAXLOCAL = 100000
MAXGLOBAL = 100000000
Matriz Transposta:
1 5 9 13
2 6 10 14
3 7 11 15
4 8 12 16
meuVetor[0] = -2
meuVetor[1] = 1
meuVetor[2] = 7
meuVetor[3] = 6
A quantidade de 1 eh 1
A quantidade de 2 eh 1
A quantidade de 3 eh 3
A quantidade de 4 eh 0
A quantidade de 5 eh 2
A quantidade de 6 eh 0
A quantidade de 7 eh 0
A quantidade de 8 eh 0
A quantidade de 9 eh 0
A quantidade de 10 eh 0
A quantidade de 11 eh 0
A quantidade de 12 eh 0
A quantidade de 13 eh 0
A quantidade de 14 eh 0
A quantidade de 15 eh 1

Análise das linhas importantes:

```cpp
int n; scanf("%d", &n);
int v[n];
for(int i = 0; i < n; i++) scanf("%d", &v[i]);
```

Neste trecho do código estamos declarando um inteiro **n** lendo seu valor, no exemplo da entrada foi **3**, e depois declaramos um vetor de tamanho **n** e então lemos **n** números e armazenamos neste vetor, no exemplo da entrada foi **5, 6, 7**. A princípio não há nada de errado em ler a quantidade de números necessária primeiro e depois criar um vetor com este tamanho, mas isto pode levar a alguns erros. Primeiro, quando declaramos **v[n]** temos exatamente **n** variáveis, e somos obrigados a usar a convenção de índices de **0** à **n - 1**, as vezes é mais conveniente usar a convenção de **1** à **n** (ou por preferência mesmo) e caso você esqueça que declarou exatamente **n** variáveis, isso pode gerar um erro de execução ao acessar o índice **n**; por

isso o recomendado é sempre declarar um pouco a mais que o necessário, por exemplo **v[n + 10]**, isso é o que chamamos informalmente de **"gordura"**, e serve para evitar erros de convenção. O segundo erro que este trecho pode gerar é exceder o limite de memória permitido dentro do escopo local, ou seja, existe um limite de memória total que pode ser alocada dentro do escopo local (por exemplo dentro do **int main()**, como é o caso), mas quando testamos nosso código em geral usamos apenas valores pequenos de **n** o que sempre funciona, mas se houver um caso de teste onde o **n** seja grande (por exemplo 10^6) o seu código pode exceder este limite de memória local. Como podemos este tipo de erro ? Acontece que existe um outro limite de memória total que pode ser alocada no escopo global (neste caso fora do **int main()**), e este limite global é bem maior que o limite local, portanto quando sabemos que um vetor pode ser grande devemos declará-lo no escopo global, mas note que declarando fora da main não temos como saber qual o valor de **n**, então iremos declarar para um valor **MAXN** que representa o maior valor de **n** possível, perceba que desta forma estaremos declarando mais que o necessário, mas isto não tem problema desde que não exceda o limite de memória. Assim a declaração ficaria desta forma:

```
const int MAXN = 1000010;
int v[MAXN];
```

Perceba que no **MAXN** já colocamos uma gordura de valor **10**. Outra vantagem de se declarar no escopo global é que toda variável declarada no escopo global começa com valor zero. Em programação competitiva é bem comum declararmos vetores grandes desta forma (também para vetores multidimensionais).

```
int vetorLocal[MAXLOCAL];
printf(" MAXLOCAL = %d\n",MAXLOCAL);
printf("MAXGLOBAL = %d\n",MAXGLOBAL);
```

Este trecho serve para mostrar a diferença entre os limites de memória local e global. Estes valores não são precisos, aqui usamos o **MAXLOCAL** como 10^5 e o **MAXGLOBAL** como 10^8 apenas para ressaltar que para vetores de tamanho total maiores que 10^5 você já deve declarar no escopo global. Apesar de não serem precisos (até porque cada problema pode ter um limite de memória diferente, o que deve ser informado no enunciado) estes números indicam bem a ordem de grandeza dos limites permitidos, por exemplo se tentar declarar um vetor no escopo global com tamanho 10^9 dará problema. Outro erro comum para iniciantes é declarar um vetor MUITO grande, por exemplo **int m[MAXLOCAL][MAXLOCAL]** que teria tamanho total de 10^{10} e isso gera erros difíceis de identificar, pois não há erro de compilação, por isso siga as recomendações para evitar tais erros.

```
int m; scanf("%d",&m);
for(int i = 0; i < m; i++)
    for(int j = 0; j < m; j++)
        scanf("%d", &matriz[i][j]);

printf("Matriz Transposta:\n");
for(int j = 0; j < m; j++) {
    for(int i = 0; i < m; i++)
        printf("%d ",matriz[i][j]);
    printf("\n");
}
```

Este trecho mortra duas formas de percorrer uma matriz, primeiramente estamos percorrendo a matriz linha por linha, e fixado uma linha passamos por todas as colunas desta linha, perceba que a variável auxiliar **i** no primeiro **for** serve para marcar em qual linha estamos, e a variável auxiliar **j** do segundo **for** para marcar em qual coluna estamos. Desta forma a ordem que passamos pela matriz no caso do exemplo 4 x 4 é a seguinte:

j\i	**0**	**1**	**2**	**3**
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

Depois estamos percorrendo a matriz coluna por coluna, e fixado uma coluna passamos por todas as linhas desta coluna, perceba que a variável auxiliar **j** no primeiro **for** serve para marcar em qual coluna estamos, e a variável auxiliar **i** do segundo **for** para marcar em qual linha estamos. Desta forma a ordem que passamos pela matriz no caso do exemplo 4 x 4 é a seguinte:

j\i	0	1	2	3
0	1	5	9	13
1	2	6	10	14
2	3	7	11	15
3	4	8	12	16

```
int meuVetor[] = {-2,1,7,6};
```

Nesta linha estamos declarando um vetor sem informar o tamanho, mas já atribuindo valores a ele, desta forma o compilador consegue calcular o tamanho necessário, neste caso é 4.

```
const int MAXVAL = 15 + 1;
int marc[MAXVAL];
...
int tam; scanf("%d", &tam);
for(int i = 0; i < tam; i++) {
    int num; scanf("%d", &num);
    marc[num]++;
}
for(int val = 1; val < MAXVAL; val++)
    printf("A quantidade de %d eh %d\n", val, marc[val]);
```

Neste trecho quero mostrar uma ideia simples porém muito útil, é o que denominamos de **vetor de marcação**, consiste em criar um vetor com o intuito de armazenar uma informação para cada índice, e quando recebemos um determinado valor **num** sua informação estará no índice **num**. No exemplo **marc** é o vetor de marcação onde estamos guardando quantas vezes cada número apareceu na sequência dada, perceba que usamos **MAXVAL = 15 + 1** apenas para exemplificar a técnica para valores de 1 à 15, mas este vetor poderia ter tamanho até 10^8 por exemplo (próximo do limite global). Assim ao ler um número **num** da sequência basta incrementar o valor de **marc[num]** para manter a quantidade de vezes que tal número apareceu, perceba que como **marc** foi declarado no escopo global ele já começa zerado. Esta ideia possui limitações, por exemplo o valor dos números devem estar entre 0 e 10^8 (não lida bem com negativos e nem com números muito grandes pois não podemos declarar um vetor tão grande), mais adiante iremos estudar estruturas que nos ajudarão a resolver estas limitações.

Lista de Exercícios:

- MINADO12
- PECA7
- Loteria
- Notas
- Tacos de Bilhar
- Botas Perdidas
- Fita Colorida
- PUSAPO11
- Quebra-Cabeça
- MINHOCA
- Torre
- Quadrado
- Robô
- PESCA11

1.5 Aula 5: Funções

Através das estruturas de repetição vimos uma forma de repetir o mesmo trecho de código várias vezes, agora veremos uma forma diferente de reutilizar o mesmo trecho de código, usando Funções. Podemos entender uma função como se fosse um pequeno programa que possui entradas e uma única saída, e este programa pode ser utilizado como um comando em outras partes do código.

Definindo uma Função

Para definir uma função usamos a seguinte sintaxe:

```
tipo nome(argumentos){
    comandos;
}
```

Conforme mencionado uma função pode possuir várias entradas, que é o que denominamos de **argumentos**, cada argumento consiste em uma variável que será usada na função e seu valor deve ser informado quando a função for utilizada. Uma função também possui uma única saída, e o **tipo** da função corresponde ao tipo da sua saída. Dentro do escopo de uma função (definido pelas chaves) definimos os comandos que serão executados toda vez que a função for utilizada, perceba que podemos criar variáveis que serão utilizadas apenas durante a execução daquela chamada da função, podemos chamar outras funções, enfim se comporta como um pequeno programa. Vale ressaltar que funções podem não ter saída, dizemos que a função não possui retorno e o tipo para este caso é o **void**; funções também podem não possuir entradas, assim basta deixar vazia a lista de argumentos vazia. Alguns exemplos de funções:

```
int maximo(int a, int b) {
    if(a >= b) return a;
    return b;
}

void imprimeDivisores(int a) {
    printf("Divisores de %d\n", a);
    for(int i = 1; i <= a; i++)
        if(a % i == 0) printf("%d\n", i);
}

void imprimeDivisoresDoMaior(int a, int b) {
    imprimeDivisores(maximo(a, b));
}
```

A primeira função possui tipo **int**, o que significa que ela retorna um valor inteiro (sua saída é um número inteiro), seu nome é **maximo**, ela recebe dois argumentos: um número inteiro **a** e um número inteiro **b**; Note que os argumentos devem ser separados por vírgula. Esta função retorna o maior valor entre **a** e **b**, perceba que o comando **return** é usado para retornar um valor e termina a chamada da função, por isso não precisamos do comando **else** neste caso, pois se entrar no **if** irá retornar **a** e terminar a função, senão consequentemente irá retornar **b**. A segunda função possui tipo **void** o que significa que não retorna nenhum valor (apesar disso pode-se utilizar o comando **return** sem nenhum um valor, apenas para terminar a chamada da função) e ela imprime todos os divisores do inteiro **a** passado como argumento. Já a terceira função utiliza as duas anteriores para imprimir os divisores do maior valor entre **a** e **b**, passados como argumentos.

Chamando uma Função

Para chamar (utilizar) uma função usamos a seguinte sintaxe:

```
nome(valor_dos_argumentos);
```

Note que na chamada da função precisamos passar o valor que será usado em cada argumento. Vale ressaltar que o valor de um argumento pode ser o retorno de uma função, é o que ocorre na função **imprimeDivisoresDoMaior** pois ela utiliza a função **maximo** para calcular o maior valor entre seus argumentos **a** e **b**, e usa este maior valor como argumento para a função **imprimeDivisores**.

É importante entender que toda vez que há uma chamada de função a execução da linha atual é pausada, inicia-se a execução

da função, e ao final seu valor de retorno é substituído no lugar onde ela foi chamada e a execução retorna exatamente onde havia parado (caso queira entender melhor como isto funciona pesquise sobre **Pilha de Execução**).

Função Recursiva

Dizemos que uma função é **Recursiva** se ela chama ela mesma, ou seja, depende dela mesma calculada para outros argumentos. Para que tal função não entre em loop infinito (fique sempre chamando ela mesma e nunca retorne) é preciso ter o que chamamos de base da recursão, ou seja, para alguns valores dos argumentos retornamos o resultado sem precisar de uma chamada recursiva, também é necessário garantir que para todos os argumentos que chamarmos a função em algum momento ela irá chamar uma das bases da recursão.

Para uma maior compreensão vamos analisar alguns códigos:

```cpp
#include <bits/stdc++.h>

const double PI = acos(-1);

double pitagoras(double cat1, double cat2) {
   return sqrt(cat1 * cat1 + cat2 * cat2);
}

int main() {
  printf("%f\n", pitagoras(3.0,4.0));

  printf("-----Funções Matemáticas-----\n");
  printf("cos(3.141593) = %f\n", cos(3.141593));
  printf("sin(3.141593) = %f\n", sin(3.141593));
  printf("tan(3.141593) = %f\n", tan(3.141593));
  printf("acos(-1) = %.14f\n", acos(-1)); // resultado de [0, pi] radianos
  printf("asin(0) = %f\n", asin(0));      // resultado de [-pi/2, pi/2] radianos
  printf("atan(0) = %f\n", atan(0));      // resultado de [-pi/2, pi/2] radianos
  printf("atan2(0, -1) = %f\n", atan2(0, -1));
  printf("atan2(0, 1) = %f\n", atan2(0, 1));
  printf("log(2.7182818) = %f\n", log(2.7182818));
  printf("log10(100) = %f\n", log10(100));
  printf("pow(2, 3) = %f\n", pow(2, 3));
  printf("sqrt(100) = %f\n", sqrt(100));
  return 0;
}
```

Saída:

```
5.000000
-----Funções Matemáticas-----
cos(3.141593) = -1.000000
sin(3.141593) = -0.000000
tan(3.141593) = 0.000000
acos(-1) = 3.14159265358979
asin(0) = 0.000000
atan(0) = 0.000000
atan2(0, -1) = 3.141593
atan2(0, 1) = 0.000000
log(2.7182818) = 1.000000
log10(100) = 2.000000
pow(2, 3) = 8.000000
sqrt(100) = 10.000000
```

Análise das linhas importantes:

```cpp
double pitagoras(double cat1, double cat2) {
   return sqrt(cat1 * cat1 + cat2 * cat2);
}
```

Aqui criamos uma função que usa o Teorema de Pitágoras para calcular o tamanho da Hipotenusa de um triângulo retângulo dados os valores dos catetos como argumentos. Note que esta função é bem curta, então poderíamos ao invés de criar esta função sempre utilizar direto $\sqrt{cat1^2 + cat2^2}$, porém dependendo do uso o código pode ficar complicado de entender (especialmente para quem não escreveu o código), perceba então que uma das utilidades de funções também é ajudar a organizar melhor o código, pois funções com bons nomes ajudam a entender melhor o código (o que ajuda quando for debugar).

```
const double PI = acos(-1);
...
printf("acos(-1) = %.14f\n", acos(-1)); // resultado de [0, pi] radianos
```

Além de mostrar o uso da função arco cosseno, queremos mostar que **acos(-1)** $= \pi$ e que desta forma podemos obter o valor de π com uma boa precisão.

```
printf("atan(0) = %f\n", atan(0));      // resultado de [-pi/2, pi/2] radianos
printf("atan2(0, -1) = %f\n", atan2(0, -1));
printf("atan2(0, 1) = %f\n", atan2(0, 1));
```

Além de mostrar o uso da função arco tangente, queremos mostrar que a função **atan(x)** retorna apenas resultados no intervalo $[-\pi/2, \pi/2]$, o que pode gerar erros em seu código, por isso recomendamos o uso da função **atan2(x, y)** que calcula o arco tangente de **x/y**, e através do sinal de **x** e **y** consegue diferenciar os arcos com o mesmo valor de tangente.

```
int main() {
    ...
    return 0;
}
```

Perceba que nós já usávamos funções anteriormente, por exemplo a própria **main** é uma função, e quando ela retorna **0** significa que na execução do programa deu tudo certo. Perceba também que **printf** e **scanf** são funções.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5    printf("-----Funções Úteis-----\n");
6    int a = 2, b = 10;
7    printf("max(%d, %d) = %d\n", a, b, max(a, b));
8    printf("min(%d, %d) = %d\n", a, b, min(a, b));
9    printf("abs(%d) = %d\n", a - b, abs(a - b));
10   swap(a, b);
11   printf("a = %d b = %d\n", a, b);
12   return 0;
13 }
```

Saída:
—–Funções Úteis—–
max(2, 10) = 10
min(2, 10) = 2
abs(-8) = 8
a = 10 b = 2

Análise das linhas importantes:

```
using namespace std;
```

Para usar as funções **max**, **min**, **swap** precisamos colocar esta linha, senão teríamos que usar o prefixo **std::** toda vez que usar alguma dessas funções.

```
swap(a, b);
```

Perceba que a função **swap(a, b)** troca os valores de **a** e **b**.

```cpp
#include <bits/stdc++.h>

int h(int a, int b, int c) {
  printf("    Chamada h(%d, %d, %d)\n", a, b, c);
  int r = 4 * (b - a) + c;
  printf("    h(%d, %d, %d) = %d\n", a, b, c, r);
  return r;
}

int g(int a, int b) {
  printf("  Chamada g(%d, %d)\n", a, b);
  int r = h(b - a, b + a, b/(a - b));
  printf("  g(%d, %d) = %d\n", a, b, r);
  return r;
}

int f(int a, int b) {
  printf("Chamada f(%d, %d)\n", a, b);

  int c = g(a, b) -2 * g(b, a);
  printf("Dentro de f: c = %d\n", c);

  int d = g(a, c) + g(b, c);
  printf("Dentro de f: d = %d\n", d);

  int r = g(c, d);
  printf("f(%d, %d) = %d\n", a, b, r);
  return r;
}

int main() {
  printf("-----Chamada de Funções-----\n");
  printf("%d\n", f(2, 1));
}
```

Saída:

```
----Chamada de Funções----
Chamada f(2, 1)
  Chamada g(2, 1)
    Chamada h(-1, 3, 1)
    h(-1, 3, 1) = 17
  g(2, 1) = 17
  Chamada g(1, 2)
    Chamada h(1, 3, -2)
    h(1, 3, -2) = 6
  g(1, 2) = 6
Dentro de f: c = 5
  Chamada g(2, 5)
    Chamada h(3, 7, -1)
    h(3, 7, -1) = 15
  g(2, 5) = 15
  Chamada g(1, 5)
    Chamada h(4, 6, -1)
    h(4, 6, -1) = 7
  g(1, 5) = 7
Dentro de f: d = 22
  Chamada g(5, 22)
    Chamada h(17, 27, -1)
    h(17, 27, -1) = 39
  g(5, 22) = 39
f(2, 1) = 39
39
```

Análise do código:

Neste código queremos mostrar com um exemplo como funciona a execução do código quando há uma chamada de função. Incluímos vários **printf**s para ajudar o entendimento, mas sugerimos que tente calcular **f(2, 1)** passo a passo e ir conferindo os valores parciais, pois entender a fundo este funcionamento é muito importante. Vale ressaltar que toda vez que uma função é chamada, é como se fosse um outro mini programa, então as varíáveis usadas são outras.

```cpp
#include <bits/stdc++.h>

int fibonacci(int n) {
  printf("Chamada fibonacci(%d)\n", n);
  if(n == 0 || n == 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
  printf("-----Função Recursiva-----\n");
  printf("%d\n", fibonacci(4));
}
```

Saída:

```
----Função Recursiva----
Chamada fibonacci(4)
Chamada fibonacci(3)
Chamada fibonacci(2)
Chamada fibonacci(1)
Chamada fibonacci(0)
Chamada fibonacci(1)
Chamada fibonacci(2)
Chamada fibonacci(1)
Chamada fibonacci(0)
3
```

Análise do código:

Neste código queremos mostrar com um exemplo como funciona uma função recursiva. Perceba que a função **fibonacci** calcula o n-ésimo número da sequência de Fibonacci, e para isso ela utiliza ela mesma para calcular o (n-1)-ésimo e o (n-2)-ésimo termos. Para quem entendeu a fundo o funcionamento do código anterior, este será fácil de entender pois é o mesmo funcionamento, por mais que seja a mesma função, como dito anteriormente cada chamada possio suas próprias variáveis então não importa se estamos chamando a mesma função ou outra o funcionamento será o mesmo. Vale lembrar que para não entrar num loop infinito, devemos definir as bases da recursão, que neste caso são os valores **0** e **1**, note que para eles não há chamada recursiva.

```cpp
#include <bits/stdc++.h>

void altera(int x, int v[], int size) {
  x++;
  printf("Agora o x = %d\n", x);

  for(int i = 0; i < size; i++) {
    v[i]++;
    printf("Agora o v[%d] = %d\n", i, v[i]);
  }
}

int main() {
  printf("---Passando Vetor como Argumento---\n");
  int meuVetor[] = {-2,1,7,6};
  int x = 10;
  altera(x, meuVetor, 4);
  printf("Após a chamada da função:\n");
  printf("x = %d\n", x);
  for(int i = 0; i < 4; i++)
    printf("meuVetor[%d] = %d\n", i, meuVetor[i]);
}
```

Saída:
—Passando Vetor como Argumento—
Agora o x = 11
Agora o v[0] = -1
Agora o v[1] = 2
Agora o v[2] = 8
Agora o v[3] = 7
Após a chamada da função:
x = 10
meuVetor[0] = -1
meuVetor[1] = 2
meuVetor[2] = 8
meuVetor[3] = 7

Análise do código:

Neste código queremos mostrar o que acontece quando passamos um vetor como argumento, e a diferença em relação a passar uma única variável. Perceba que a variável **x** dentro da função **main** foi inicializada com o valor **10**, passamos ela para a função **altera** que incrementou este valor virando **11**, mas após a chamada da função **altera** note que o valor da variável **x** continuou **10**, porque na verdade quando passamos uma variável para uma função na realidade é criada uma cópia desta variável, então a variável **x** dentro da chamada da função **altera** não é a mesma variável **x** da função **main**, são duas variáveis diferentes que no momento que chamamos a função **altera** os valores delas são iguais pois uma é copiada da outra, então quando fazemos **x++** na função **altera** estamos alterando apenas a variável **x** da função **altera** e a variável **x** da função **main** permanece inalterada. Porém quando lidamos com vetores, o funcionamento é diferente, note que o vetor **meuVetor** de fato foi alterado. O motivo desta diferença ficará claro quando estudarmos Ponteiros.

Lista de Exercícios:

- F91
- Fibonacci, Quantas Chamadas?
- Brick Wall Patterns
- ACOES1MG
- FEYNMAN
- Meme Wars

1.6 Aula 6: String

Uma string é simplesmente uma sequência de caracteres, por exemplo palavras são consideradas strings, e para lidar com uma string usamos um vetor de caracteres. Existem alguns detalhes importantes qua precisamos saber para manipular strings, é o que veremos a serguir. Vale ressaltar que agora iremos ver como tratar strings de acordo com a linguagem C, o que também funciona na linguagem C++, porém em C++ existe uma estrutura de dados chamada **string** que facilita muitas operações, mas só estudaremos esta estrutura quando falarmos sobre STL.

Estrutura

Conforme dito anteriormente usaremos um vetor de caracteres para armazenar uma string, mas existe um detalhe importante, o final da string é marcado com o caractere **'\0'** (ele é auxiliar apenas para mostrar que a string terminou) que possui valor na tabela ASCII igual a zero, por exemplo para armazenar a string **'VTEX'** precisaremos de tamanho 5:

'V'	'T'	'E'	'X'	'\0'
0	1	2	3	4

Leitura

Para ler uma string usamos o comando **scanf** com o caractere especial **%s**, da seguinte forma:

```
scanf(" %s", nome);
```

Alguns detalhes importantes são:

- O comando **scanf** lê até encontrar um caractere em branco na entrada (por exemplo um espaço, um tab, uma quebra de linha, entre outros).
- Note que usamos um espaço antes do caracter especial para string **␣%s**, este espaço pula todo caractere em branco antes de um caractere válido da entrada e assim evita que você leia uma string vazia.
- Perceba que não usamos o caractere **&**, isso também ficará mais claro quando estudarmos Ponteiros.
- O comando **scanf** já coloca automaticamente o caractere **'\0'** no final da string, então lembre-se de declarar o vetor com um tamanho que funcione (outro motivo para declarar um vetor com gordura).

Algumas vezes queremos ler uma linha inteira como uma string, mesmo que ela tenha espaços no meio, assim o comando acima não funcionará pois irá parar no primeiro caractere em branco (após o primeiro não branco por conta do espaço), o comando para esta situação é o seguinte:

```
scanf(" %[^\n]", linha);
```

Impressão

Para imprimir uma string usamos o seguinte comando:

```
printf("%s\n", nome);
```

O comando **printf** identifica o final da string através do caractere **'\0'**.

Funções Úteis

Existem algumas funções que ajudam bastante a manipular strings, são elas:

- **String Length:** esta função retorna o tamanho da string, ou seja, a quantidade de caracteres antes do **'\0'** (ele não é contado)

  ```
  int tam = strlen(nome);
  ```

- **String Compare:** frequentemente queremos comparar duas strings lexicograficamente (por exemplo qual palavra vem antes alfabeticamente) e infelizmente não podemos usar **s1 < s2** pois o operador **<** não está definido para vetor de char. Para isso usamos a função **strcmp(s1, s2)** que irá retornar um valor **negativo** caso **s1** seja menor, **zero** caso as strings sejam iguais, ou **positivo** caso **s2** seja menor.

- **String Copy:** frequentemente queremos copiar o valor de uma string para outra, mas infelizmente não podemos usar **s1 = s2** pois o operador de atribuição não está definido para vetor de char. Para isso usamos a função **strcpy(s1, s2)** que copia o valor de **s2** para **s1**. Observe que a string **s1** deve ter espaço suficiente para caber o valor de **s2**, e que o valor de **s2** não é alterado.

Para uma maior compreensão vamos analisar o seguinte código:

```
#include <bits/stdc++.h>

char linha[100];

int main() {
  char s[] = "Minha String";
  printf("%s\n", s);
  int tam = strlen(s);
  printf("último caractere=%c número=%d\n",
   ↪  s[tam], s[tam]);

  s[3] = 0;
  printf("Novo tamanho: %d\n", strlen(s));

  char nome1[15], nome2[15];
  scanf(" %s %s", nome1, nome2);
  printf("nome1 = %s\n", nome1);
  printf("nome2 = %s\n", nome2);

  if(strcmp(nome1, nome2) < 0)
    printf("%s < %s\n", nome1, nome2);
  else
    printf("%s < %s\n", nome2, nome1);

  scanf(" %[^\n]", linha);
  printf("%s\n", linha);

  strcpy(linha + strlen(linha), nome1);
  strcpy(linha + strlen(linha), nome2);
  printf("Nova Linha:\n%s\n", linha);
}
```

Entrada:
André Athos
Colaboradores deste material:
Saída:
Minha String
último caractere= número=0
Novo tamanho: 3
nome1 = André
nome2 = Athos
André < Athos
Colaboradores deste material:
Nova Linha:
Colaboradores deste material:AndréAthos

Análise das linhas importantes:

```
char s[] = "Minha String";
printf("%s\n", s);
int tam = strlen(s);
printf("último caractere=%c número=%d\n", s[tam], s[tam]);
```

Neste trecho temos a declaração de uma string usando aspas duplas (lembre-se que aspas simples é usada para caracteres), e também mostramos que o caractere '\0' é um caractere em branco pois quando imprimimos ele com **%c** sai um espaço em branco, e seu valor na tabela ASCII é 0, o que pode ser verificado quando imprimimos com **%d**.

```
s[3] = 0;
printf("Novo tamanho: %d\n", strlen(s));
```

Aqui queremos mostrar que se você alterar algum caractere de uma string para o valor 0, ou para o caractere '\0', então é este será o novo final, assim apesar de termos outros caracteres após o índice 3, eles serão ignorados pois o '\0' indica o final

da string.

```
char nome1[15], nome2[15];
scanf(" %s %s", nome1, nome2);
printf("nome1 = %s\n", nome1);
printf("nome2 = %s\n", nome2);

if(strcmp(nome1, nome2) < 0)
    printf("%s < %s\n", nome1, nome2);
else
    printf("%s < %s\n", nome2, nome1);
```

Neste exemplo mostramos como ler palavra por palavra da entrada, se atente aos espaços para pular os caracteres em branco, e também mostramos como usar a função **strcmp** para comparar duas strings lexicograficamente.

```
scanf(" %[^\n]", linha);
printf("%s\n", linha);

strcpy(linha + strlen(linha), nome1);
strcpy(linha + strlen(linha), nome2);
printf("Nova Linha:\n%s\n", linha);
```

Neste trecho temos um exemplo de como ler uma linha inteira (mesmo com espaços) e depois utilizando a função **strcpy** estamos concatenando (inserindo no final) as strings **nome1** e **nome2** lidas anteriormente.

Lista de Exercícios:

- TEL8
- JVESTI08
- Decifra
- Translation
- Pangram
- String Task
- k-String
- Bear and Strings
- Letras
- String Cutting

1.7 Aula 7: Struct

Quando estudamos variáveis vimos os tipos básicos (por exemplo: **int**, **char**, **double**, etc), agora usando os tipos básicos veremos como criar nossos próprios tipos, através do que chamamos de **Struct**.

Definição

Para declarar uma nova struct usamos a seguinte sintaxe:

```
struct nome {
    tipos constituíntes;
};
```

Onde **nome** é o nome que damos a esta struct e **tipos constituíntes** são as variváeis que farão parte desta struct, por exemplo:

```
struct ponto {
  int x, y;
};
```

A struct **ponto** possui dois inteiros dentro dela, denominados de **x** e **y**. Não se esqueça do **;** ao final da declaração da struct.

Declaração

Para declarar uma variável cujo tipo seja uma struct previamente definida usamos a mesma sintaxe que declarar uma variável de um tipo básico, o que muda é o nome do tipo, por exemplo:

```
ponto a;
```

Aqui estamos declarando uma variável do tipo **ponto** com nome **a**.

Acesso

Para acessar as variáveis que constituem uma struct usamos **.**, por exemplo:

```
a.x
```

Aqui estamos acessando a variável **x** dentro da struct **a**, que é do tipo **int** e se comporta como qualquer outro **int**, ou seja podemos usar como usamos um **int** qualquer.

Métodos

Também é possível definir funções dentro structs, na verdade elas são chamadas de métodos da struct, por exemplo:

```
struct ponto {
  int x, y;
  int soma() {
    return x + y;
  }
};
```

Aqui estamos definindo o método **soma** da struct **ponto**, e para chamar este método também usamos **.**, por exemplo:

```
a.soma()
```

A linguagem C++ é orientada à Objetos, por conta disso existem muitas outras funcionalidades de **struct** e **class**, mas para Programação Competitiva estas funcionalidades geralmente não são necessárias e por isso não iremos entrar em detalhes, porém caso queira se aprofundar em C++ recomendamos seguir os tutoriais do cplusplus (especialmente os de Classes).

Para uma maior compreensão vamos analisar os seguintes códigos:

```cpp
#include <bits/stdc++.h>

#define SEPARADOR "------------------------------"

struct data {
  int dia, mês, ano;

  void imprime() {
    printf("%d/%d/%d\n", dia, mês, ano);
  }
};

struct aluno {
  char nome[100];
  int idade;
  data nascimento;

  void imprime() {
    printf("Nome Completo: %s\n", nome);
    printf("Idade: %d\n", idade);
    printf("Data de Nascimento: ");
    nascimento.imprime();
  }
};

int main() {
  aluno alunos[] = {
    {"Arnaldo Primeiro", 25, {9, 2, 1994}},
    {"Bernaldo Segundo", 26, {10, 3, 1993}},
    {"Cernaldo Terceiro", 27, {11, 4, 1992}},
  };

  for(int i = 0; i < 3; i++) {
    if(i > 0) printf("%s\n", SEPARADOR);
    alunos[i].imprime();
  }

  return 0;
}
```

Saída:

```
Nome Completo: Arnaldo Primeiro
Idade: 25
Data de Nascimento: 9/2/1994
------------------------------
Nome Completo: Bernaldo Segundo
Idade: 26
Data de Nascimento: 10/3/1993
------------------------------
Nome Completo: Cernaldo Terceiro
Idade: 27
Data de Nascimento: 11/4/1992
```

Análise das linhas importantes:

```cpp
#define SEPARADOR "------------------------------"
```

Aqui estamos definindo uma macro usando o seguinte comando: **#define identificador valor**, ela é processada antes da compilação e basicamente substitui o **identificador** pelo **valor** em todo lugar que ele aparece. Muitas vezes é possível evitar o uso do **#define** e usar uma variável **const**.

```cpp
aluno alunos[] = {
    {"Arnaldo Primeiro", 25, {9, 2, 1994}},
    {"Bernaldo Segundo", 26, {10, 3, 1993}},
    {"Cernaldo Terceiro", 27, {11, 4, 1992}},
};
```

Perceba que podemos mesclar a vontade todos os conceitos que apendemos, assim estamos usando um vetor de struct. Note também que a struct **aluno** utiliza a struct **data**.

```c
#include <bits/stdc++.h>

struct str1 {
  char a;
  char b;
  int c;
};

struct str2 {
  char a;
  int c;
  char b;
};

int main() {
  printf("size of char: %d\n", sizeof(char));
  printf("size of int: %d\n", sizeof(int));

  int v[5];
  printf("size of v: %d\n", sizeof(v));
  printf("tamanho de v: %d\n",
  ↪  sizeof(v)/sizeof(int));

  printf("size of str1: %d\n", sizeof(str1));
  printf("size of str2: %d\n", sizeof(str2));

  return 0;
}
```

Saída:
size of char: 1
size of int: 4
size of v: 20
tamanho de v: 5
size of str1: 8
size of str2: 12

Análise das linhas importantes:

```c
printf("size of char: %d\n", sizeof(char));
printf("size of int: %d\n", sizeof(int));
```

Aqui estamos usando o comando **sizeof(tipo)** que neste caso retorna o tamanho do **tipo** em bytes, note que o tamanho do **char** deu 1 e do **int** deu 4 (condizente com o que aprendemos anteriormente).

```c
int v[5];
printf("size of v: %d\n", sizeof(v));
printf("tamanho de v: %d\n", sizeof(v)/sizeof(int));
```

O comando **sizeof** também calcula o tamanho em bytes de um bloco de memória (passando o ponteiro para o início do bloco), neste caso estamos calculando o tamanho em bytes do vetor **v**. Note que o tamanho de **v** é 20 bytes, pois ele é composto de 5 **int**s e cada **int** tem tamanho de 4 bytes. Também mostramos uma forma de encontrar o tamanho de um vetor utilizando o comando **sizeof**, basta dividir o seu tamamnho em bytes pelo tamanho do seu tipo em bytes.

```c
struct str1 {
  char a;
  char b;
  int c;
};

struct str2 {
  char a;
  int c;
  char b;
```

```
};
...
printf("size of str1: %d\n", sizeof(str1));
printf("size of str2: %d\n", sizeof(str2));
```

Neste trecho perceba que apesar de as structs **str1** e **str2** possuírem os mesmos tipos constituintes elas possuem tamanhos diferentes, isso ocorre devido às restrições de alinhamento de memória, não entraremos em detalhes (caso queira entender melhor: Stack Overflow) mas o importante é que dependendo do uso structs podem gastar mais memória do que deveriam, tome cuidado e teste !

1.8 Aula 8: Ponteiros e Referências

Muitas vezes queremos que uma função altere o valor de uma variável, mas conforme vimos anteriormente ao passar uma variável para uma função estamos na verdade passando uma cópia. uma forma de conseguir isso é declarando a variável em questão no escopo global (fora de todas as funções) assim ela fica acessível a todas as funções e portanto todas podem alterá-la. Esta forma é simples e muito utilizada em programação competitiva, mas em termos de desenvolvimento de software esta prática é bem ruim (por exemplo porque dá acesso a mais funções do que deveria) e também o código de algumas estruturas de dados usadas em programação competitiva frequentemente utilizam a forma correta, por isso nesta aula estudaremos formas de dar acesso a uma variável, ou seja, permitir que uma função altere uma variável mesmo ela não tendo sido declarada nesta função.

Ponteiros

Ponteiros são variáveis que guardam endereços de outras variáveis, ou seja, o valor de um ponteiro é o local da memória onde uma outra variável está armazenada, dizemos que o ponteiro 'aponta' para esta outra variável.

Declaração de um Ponteiro

Para declarar um ponteiro usamos a seguinte sintaxe:

```
tipo *nome;
```

Onde **tipo** indica o tipo de variável que este ponteiro pode apontar. Por exemplo:

```
int *p;
```

Aqui **p** é um ponteiro para uma variável do tipo **int**. É importante dizer que o tipo de **p** é **int ***. Existe o tipo **void *** que permite que o ponteiro aponte para qualquer posição da memória.

Atribuição para um Ponteiro

Para obter o endereço de uma variável basta usar o **&**, e assim atribuir ao ponteiro, por exemplo:

```
int a;
int *p = &a;
```

Aqui estamos atribuindo o endereço da variável **a** ao ponteiro **p**, assim dizemos que **p** aponta para **a**.

Acesso usando um Ponteiro

Para acessar a variável que um ponteiro acessa usamos a seguinte sintaxe:

```
*nome
```

Tome cuidado para não confundir este ***** com o que está presente na declaração do ponteiro, quando usamos **tipo *nome** o asterísco faz parte do tipo do ponteiro, aqui o asterísco serve para acessar a variável que o ponteiro aponta, isto também é chamado de dereferenciar o ponteiro. Desta forma se temos um ponteiro para uma variável podemos através do ***** acessar essa variável e alterar/usar como quisermos, por exemplo:

```
int a = 10;
int *p = &a;
*p = 20;
```

Neste exemplo alteramos o valor da variável **a** utilizando um ponteiro.

Referências

Uma Referência é um pseudônimo, ou seja, um outro nome para uma variável que já existe. Ela não ocupa espaço na memória pois se trata da mesma variável já existente, mas ela ocupa espaço na pilha de execução.

Declaração de uma Referência

Para declarar uma referência a uma variável usamos a seguinte sintaxe:

```
tipo &nome = nomeDaVariável;
```

Existem alguns cuidados necessários, primeiro que **tipo** deve ser o mesmo tipo da variável **nomeDaVariável**, segundo que a variável **nomeDaVariável** deve ter sido declarada antes da declaração da referência, e por último toda vez que declararmos uma variável temos que atribuir uma variável a ela. Por exemplo:

```
int a;
int &r = a;
```

Neste exemplo **r** é uma referência a variável **a**.

Acesso usando uma Referência

Para acessar uma variável usando uma referência, basta usá-la como se fosse a própria variável, por exemplo:

```
int a = 10;
int &r = a;
r = 20;
```

Aqui alteramos o valor da variável **a** usando a referência **r**, é como se **a** e **r** fossem a mesma variável.

Diferenças entre Ponteiro e Referência

É muito comum a confusão entre ponteiro e referência, e muitas vezes não fica claro o que cada um é exatamente, assim para um entendimento mais aprofundado iremos explorar as diferenças entre eles:

- **Declaração:** todos os cuidados que existem na declaração de uma referência não se aplicam a declaração de um ponteiro, assim podemos por exemplo declarar um ponteiro antes da declaração de uma variável e depois apenas atribuir seu endereço ao ponteiro.
- **Reatribuição:** podemos reatribuir um ponteiro conforme desejarmos (aponta para uma variável, depois aponta para outra, enfim), mas não podemos reatribuir uma referência, ou seja, uma vez que atribuímos uma variável na declaração de uma referência, não podemos mais mudar (de fato é como se a referência e a variável fossem uma mesma variável dali em diante).
- **Memória:** conforme dito anteriormente um ponteiro tem seu próprio espaço na memória, enquanto que a referência não possui espaço na memória (quem tem espaço na memória é a variável que a referência está associada). Mas tanto ponteiros quanto referências ocupam espaço na Pilha de Execução.
- **Valor Nulo:** um ponteiro pode apontar para nenhum lugar, basta atribuir seu valor para **NULL**, mas uma referência sempre deve estar associada a uma variável (atribuída na própria declaração).
- **Ponteiro de Ponteiro:** é possível ter um ponteiro apontando para outro ponteiro, é o que chamamos de ponteiro de ponteiro, por exemplo **int **p** é um ponteiro de ponteiro para inteiro. E isso pode se extender, por exemplo **int ***q** é um ponteiro de ponteiro de ponteiro para inteiro. já com referência não existe este comportamento, ou seja, não podemos criar uma referência de uma referência.
- **Operações Aritméticas:** é possível usar ponteiros em operações aritméticas, por exemplo se **p** é um ponteiro, **p + 1** é um ponteiro que aponta para a posição de memória logo em seguida da que **p** aponta. Mas não podemos usar referência em operações aritméticas.

Para uma maior compreensão vamos analisar os seguintes códigos:

```cpp
#include <bits/stdc++.h>

void modifica1(int *ptr) {
  *ptr = 20;
}

void modifica2(int &a) {
  a = 30;
}

int main() {
  int a = 10;

  int *p = &a;
  modifica1(p);
  printf("a = %d\n", a);

  int &r = a;
  modifica2(r);
  printf("a = %d\n", a);

  return 0;
}
```

Saída:
a = 20
a = 30

Análise do código:

Aqui estamos mostrando como permitir que uma função altere uma variável mesmo ela não tendo sido declarada nesta função, que foi a motivação desta aula. Perceba que fizemos de duas formas, uma utilizando ponteiro, e outra utilizando referência, e as duas são válidas. Perceba que ao passar um ponteiro ou referência para uma função, na verdade estamos passando uma cópia, mas neste caso não tem problema pois por exemplo se temos uma cópia de um ponteiro que aponta para o mesmo lugar que o original, quando alterarmos o valor da variável apontada, iremos alterar o valor da variável original.

```cpp
#include <bits/stdc++.h>

int main() {
  int *p = NULL;
  int a = 10;
  p = &a;
  int &r = a;

  printf("sizeof p = %d\n", sizeof(p));
  printf("sizeof r = %d\n", sizeof(r));

  printf("valor de a    = %d\n", a);
  printf("endereço de a = %llu\n", &a);
  printf("valor de p    = %llu\n", p);
  printf("endereço de p = %llu\n", &p);
  printf("valor de r    = %d\n", r);
  printf("endereço de r = %llu\n", &r);
  return 0;
}
```

Saída:
sizeof p = 8
sizeof r = 4
valor de a = 10
endereço de a = 140732832118428
valor de p = 140732832118428
endereço de p = 140732832118432
valor de r = 10
endereço de r = 140732832118428

Análise das linhas importantes:

```
printf("sizeof p = %d\n", sizeof(p));
printf("sizeof r = %d\n", sizeof(r));
```

Utilizando a função **sizeof** encontramos o tamanho em bytes de ponteiro **p** e da referência **r**. Todo ponteiro possui tamanho 8 bytes pois precisa armazenar um endereço da memória (perceba que isso independe do tipo da variável apontada). Já em relação a referência o **sizeof** pega o tamanho da variável que ela está associada (pois é como se as duas fossem uma só).

```
printf("valor de a    = %d\n", a);
printf("endereço de a = %llu\n", &a);
printf("valor de p    = %llu\n", p);
printf("endereço de p = %llu\n", &p);
printf("valor de r    = %d\n", r);
printf("endereço de r = %llu\n", &r);
```

Aqui estamos imprimindo tanto o valor quanto o endereço na memória de **a**, **p** e **r**. Perceba que o valor de **r** é o mesmo que o valor de **a**, que o endereço de **r** é o mesmo que o endereço de **a**, o valor de **p** é o endereço de **a**, e o endereço de **p** é diferente do endereço de **a**. Vale ressaltar que os endereços na saída devem mudar sempre que o código for executado novamente, pois a cada momento estamos alocando outras variáveis e em posições diferentes na memória.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5    int v[] = {1, 2};
6    printf("%llu %llu\n", &v[0], &v[1]);
7    printf("%llu %llu\n", v, v + 1);
8
9    int a = 10, b = 20;
10   int *pa, *pb;
11   pa = &a; pb = &b;
12   swap(pa, pb);
13   *pa = 100; *pb = 200;
14   printf("a = %d, b = %d\n", a, b);
15
16   return 0;
17 }
```

Saída:
140732820104864 140732820104868
140732820104864 140732820104868
a = 200, b = 100

Análise das linhas importantes:

```
int v[] = {1, 2};
printf("%llu %llu\n", &v[0], &v[1]);
printf("%llu %llu\n", v, v + 1);
```

Na aula sobre funções mostramos que quando passamos um vetor para uma função e alteramos o vetor dentro da função, o vetor original realmente se altera, aqui estamos mostrando o motivo. Na verdade um vetor é um ponteiro para a primeira posição do vetor, note no exemplo que o valor de **v** é o mesmo que o endereço de **v[0]**, e toda vez que acessamos uma posição do vetor estamos usando operações aritméticas com o ponteiro para a posição inicial, por exemplo num vetor de uma dimensão podemos dizer que **t[i]** equivale a ***(t + i)**, note também que no exemplo o valor de **v + 1** é o mesmo que o endereço de **v[1]**. Portanto quando passamos um vetor para uma função, estamos na verdade passando um ponteiro, e por isso conseguimos alterar o vetor.

```
int a = 10, b = 20;
int *pa, *pb;
pa = &a; pb = &b;
swap(pa, pb);
*pa = 100; *pb = 200;
printf("a = %d, b = %d\n", a, b);
```

Neste trecho estamos declarando duas variáveis, **a** e **b** com valores iniciais **10** e **20** respectivamente, depois declaramos dois ponteiros para inteiro, **pa** e **pb**, note a sintaxe para declarar mais de um ponteiro na mesma linha (se fizer **int * pa, pb;** então **pb** será apenas um **int** ao invés de um ponteiro), e então apontamos **pa** para **a** e **pb** para **b**. Em seguida usamos a função **swap** que troca o conteúdo das variáveis passadas, e como passamos **pa** e **pb**, então agora **pa** aponta para **b** e **pb** aponta para **a**, o que pode ser constatado após alterarmos os valores das variáveis que os ponteiros apontam.