



Apostila de Introdução a C/C++

**Victor Agnez Lima, Luiz Fernando Carbonera Filho,
Lucas Torres Marques, Felipe Fernandes Lopes,
Rennan Fabrício Campos Bezerra.**



Introdução

A IEEE *Computer Society* (CS) UFRN é uma entidade estudantil que tem como propósito disseminar a computação na universidade. Tendo como foco atividades nas áreas de *Internet of Things* (IoT), aprendizado de máquina e programação. Desenvolvendo também atividades multidisciplinares como por exemplo em arquitetura e com educação para crianças e adolescentes.

Agora que nos apresentamos, queremos falar um pouco sobre nossa apostila e que objetivos pretendemos alcançar com ela. A ideia de uma apostila desse tipo é para te auxiliar na sua entrada no nosso mundo, o mundo da computação, da programação no geral e te mostrar que programar não é um bicho de sete cabeças, mas sim um desafio pessoal e que se torna muito divertido conforme vamos aprendendo.

Estaremos tratando de diversos tópicos da programação, porém, na nossa apostila estaremos trabalhando especificamente com a linguagem C/C++. Essa linguagem a primeira vista pode parecer meio esquisita, mas vamos te mostrar o quanto ela é simples e o quanto você pode fazer apenas com o básico. Trataremos os assuntos de suas partes iniciais com exemplos básicos e com conceitos entregues a você de maneira mais fácil.

Ao desenvolvermos essa apostila, tentamos nos colocar no seu papel, para assim criar um conteúdo intuitivo e de fácil absorção. Ficamos felizes em te entregar esse material e esperamos que você possa aprender e, logo mais se apaixonar por esse mundo incrível e cheio de descobertas a frente. Nesse minicurso queremos desenvolver suas capacidades e demonstrar a você que você pode sim, aprender a programar seja lá para qual finalidade você pretende, enfim, chega de papo. Vamos lá?

Sumário

Introdução	i
Sumário	ii
I Aprendendo Sobre Programação	1
1 Lógica de Programação	3
1.1 Algoritmos	3
1.2 Conceitos e regras básicas de programação	4
1.3 Estrutura de decisão e repetição	4
1.4 Vetores	5
1.5 Funções	6
1.6 Bibliotecas	6
II Aprendendo a Programar em C/C++	7
1 Tipos de dados básicos em C++	9
1.1 Conversão de dados em C/C++	12
1.2 Entrada e Saída padrão de dados	15
2 Vetores (Arrays)	19
2.1 Vetores Unidimensional	19
2.2 Vetores com mais de uma dimensão (Matrizes)	21
3 Estruturas de decisão	23
3.1 Definição e condições básicas	23
3.2 Comando If-Else	23
4 Estruturas de repetição	27
4.1 Comando while	27
4.2 Comando for	28
5 Funções	31
5.1 Procedimentos	33
5.2 Arrays como argumento de uma função ou procedimento	33
6 Bibliotecas em C/C++	37
6.1 Principais bibliotecas	37
6.2 Como criar e utilizar a sua biblioteca própria	38

III Desafio Final	43
1 O famoso e icônico Jogo da Velha	45
Bibliografia	47

PARTE I

Aprendendo Sobre Programação

CAPÍTULO 1

Lógica de Programação

sec:log

1.1 Algoritmos

Para começar a nossa aula sobre lógica de programação, gostaríamos de falar um pouco sobre o conceito de **algoritmos**, uma palavrinha que muitas vezes é confundida com **logaritmos** mas que na verdade não tem nada a ver uma coisa com a outra.

O conceito de algoritmo é muito simples, nada mais é que **uma sequência de passos lógicos** que, na programação, deverá ser dada como instruções para uma máquina a fim de que ela possa realizar alguma tarefa.

Imagine que você está acordando, como seria a sequência de passos do que você faz até sair de casa? Bem, é evidente que isso depende muito de pessoa para pessoa, mas podemos dizer que uma possibilidade de algoritmo seria:

Início

- ABRIR OS OLHOS;
- LEVANTAR DA CAMA;
- ANDAR ATÉ O BANHEIRO;
- FAZER XIXI;
- LAVAR AS MÃOS;
- ESCOVAR OS DENTES;
- ANDAR ATÉ A COZINHA;
- FAZER O CAFÉ DA MANHÃ;
- COMER O CAFÉ DA MANHÃ;
- IR ATÉ O BANHEIRO;
- ESCOVAR OS DENTES;
- IR ATÉ A PORTA;
- SAIR DE CASA;

Fim

Essa sequência de passos, como já discutimos antes, pode ser chamada de Algoritmo do Acordar. Porém, vale deixar claro que um algoritmo aumenta de complexidade de acordo com a quantidade de passos a serem feitos e dificuldade da tarefa a ser feita.

1. Lógica de Programação

Agora que sabemos o que é um algoritmo, precisamos saber como nos comunicar com o computador para que ele realize os passos lógicos que nós queremos. Para isso, temos que utilizar uma linguagem que ele entenda, pois assim como na comunicação entre pessoas, a comunicação com o computador só acontece quando ambos estão usando uma linguagem conhecida tanto pelo emissor da mensagem (você, programador), quanto pelo receptor (o computador ou outra máquina que esteja programando). Nesse minicurso iremos aprender apenas duas das inúmeras linguagens que podem ser reconhecidas pelo seu computador, **C** e **C++**.

1.2 Conceitos e regras básicas de programação

Provavelmente você já ouviu falar em variáveis durante os seus estudos. Assim como na matemática, quando falamos em variável em programação, estamos nos referindo a alguma coisa que pode mudar. Na programação ela é armazenada em um espaço de memória que terá um determinado nome definido por nós. Por exemplo, suponha que você quer armazenar alguns dados a respeito de você mesmo, como o seu *nome*, a sua *idade*, o seu *salário* e o seu *sexo*. Perceba que as 4 variáveis irão armazenar 4 informações diferentes ao seu respeito, e que cada um tem um determinado valor que poderá ser mudado de acordo com a sua vontade ou necessidade.

Para o computador, cada variável dessas são dados, e dados podem assumir diversos formatos. Pense da seguinte forma:

- **Nome** é uma sequência de caracteres, como por exemplo "Lucas";
- **Idade** é um número inteiro, como por exemplo "19";
- **Salário** é um número real, como por exemplo "450,87";
- **Sexo** é algo binário, ou é "Masculino", ou é "Feminino";

Em programação, dizemos que uma variável que armazena uma sequência de caracteres é do tipo **texto**, uma variável que armazena um número inteiro é do tipo **inteiro**, uma variável que armazena um número real é do tipo **flutuante** e uma variável que armazena um **tipo de dados lógico (binária)**, como verdadeiro ou falso são chamados de **booleano**.

Todas essas variáveis podem possuir um nome ou declaração específica dependendo da linguagem. Mais a frente veremos um pouco mais sobre elas no capítulo de **Tipos de dados básicos em C++**.

Além disso, em alguns momentos na sua vida de programador você precisará transformar uma variável de um tipo para outro, isso será discutido em nossa apostila no capítulo **Conversão de dados em C/C++**.

1.3 Estrutura de decisão e repetição

Dois outros conceitos muito importantes para a programação são o de Estruturas de Decisão e Estrutura de Repetição. Uma estrutura de decisão é aquela que irá testar uma condição e, caso ela seja satisfeita, partirá para um outro bloco de sequência de código. Para ilustrar uma estrutura de decisão, vamos utilizar as palavras **SE**, **SENÃO SE** e **SENÃO**. Vajamos um exemplo:

Início

- 1 ABRIR OS OLHOS;
- 2 LEVANTAR DA CAMA;
- 3 ANDAR ATÉ O BANHEIRO;
- 4 **SE** ESTIVER APERTADO:
 - 4.1 IR AO BANHEIRO;
 - 4.2 LAVAR AS MÃOS;

- 5 ESCOVAR OS DENTES;
- 6 SE ESTIVER COM FOME:
 - 6.1 ANDAR ATÉ A COZINHA;
 - 6.2 SE TIVER COMIDA:
 - 6.3.1 COMER O CAFÉ DA MANHÃ FEITO;
 - 6.3 SENÃO:
 - 6.3.1 FAZER O CAFÉ DA MANHÃ;
 - 6.3.2 COMER O CAFÉ DA MANHÃ QUE VOCÊ FEZ;
- 7 IR ATÉ O BANHEIRO;
- 8 ESCOVAR OS DENTES;
- 9 IR ATÉ A PORTA;
- 10 SAIR DE CASA;

Fim

Agora que você já está expert em estruturas de decisões, vamos falar sobre estruturas de repetição. Uma estrutura de repetição nada mais é que uma função cuja finalidade é fazer com que um bloco de passos lógicos se repita até que uma condição definida por você deixe de ser atendida. Algumas palavras utilizadas para iniciar uma estrutura de repetição são: ENQUANTO e FAÇA (...) ENQUANTO. Por exemplo, vejamos o seguinte algoritmo:

Início

- 1 ENQUANTO A VARIÁVEL *nota_da_prova* FOR *menor que 10* FAÇA:
 - 1.1 ESTUDAR MAIS;
 - 1.2 FAÇA A PROVA;
 - 1.3 ARMAZENE A NOTA DA PROVA NA VARIÁVEL *nota_da_prova*;
2. PARABÉNS, VOCÊ SE FORMOU EM ENGENHARIA!;

Fim

1.4 Vetores

Agora que você sabe o que é uma variável, que cada variável tem um nome e que armazena um valor com determinado tipo de dado, imagine a seguinte situação: você está desenvolvendo um sistema para armazenar as notas dos alunos de uma grande escola da sua cidade. Com o conceitos que temos agora teríamos que fazer um grande conjunto de variáveis chamados, por exemplo, de *nota_aluno1*, *nota_aluno2*, *nota_aluno3* e assim sucessivamente.

Pensando nisso, em programação é possível criar uma única variável e dizer - ou como é chamado normalmente por nós programadores - declarar que ela não ocupará apenas um único espaço na memória, mas sim vários. Esse tipo de variável é chamada de vetor, ou em inglês *array*.

1.5 Funções

Existe também um conceito muito importante utilizado por todos os programadores, o de funções. De maneira geral uma função é utilizada para agrupar um conjunto de ações que são feitas várias vezes durante o código e para facilitar o entendimento do funcionamento do código.

Para entender a situação, pense que você e um amigo estão esperando o ônibus e enquanto ele está estudando você está atento a chegada do ônibus. O seu amigo poderia toda vez que quisesse saber se o ônibus estava chegando falar: "você poderia olhar na direção contrária da via, procurar um ônibus vindo e verificar se é o nosso, caso não seja pegue o seu celular, abra o aplicativo da empresa de ônibus, procure o nosso ônibus, veja a localização dele e depois me diga o resultado". Ou simplesmente perguntar "o ônibus está chegando?". No caso *"o ônibus está chegando?"* é um ótimo exemplo de como uma função pode ser usada, simplificando todas as falas ao agrupá-las, mas mesmo assim solicitando ao receptor da mensagem que execute todas as ações para verificar se o transporte está a caminho.

1.6 Bibliotecas

Agora que você já sabe o que é uma função, considere que você está trabalhando em uma empresa de desenvolvimento de softwares e percebe que toda vez que está criando um novo programa tem que ficar recriando um mesmo conjunto de funções para, por exemplo, criar o login e registro do usuário. Para otimizar esse tipo de coisa e você não precisar ficar copiando e colando as funções em cada projeto existem as **bibliotecas**.

A principal função de uma biblioteca na programação é armazenar funções, classes, métodos ou variáveis - de acordo com a necessidade do programador - que podem ser usadas de forma independente por qualquer código principal, para isso, basta dizermos ao compilador para incluir aquela biblioteca no nosso arquivo/função principal.

Na próxima parte da apostila iremos mergulhar no universo do C/C++, aplicando todos os conceitos aprendidos até o momento em situações reais e até cotidianas para um programador de C/C++, bem como introduzindo novos conceitos que são considerados essenciais para todos que querem se tornar programadores nessa linguagem. Vejo você no próximo capítulo!

PARTE II

Aprendendo a Programar em C/C++

CAPÍTULO 1

Tipos de dados básicos em C++

sec:tipd

Imagine que você vai fazer um programa para calcular a média de três idades, vamos dizer que as idades são a sua e de mais dois amigos. Para isso você precisa das três idades, da soma delas e o resultado dividido por três. Matematicamente teríamos isso:

$$media = \frac{sua_idade + idade1 + idade2}{3}. \quad (1.1)$$

Sendo *sua_idade* a idade de quem está calculando, *idade1* e *idade2* a dos seus amigos e *media* a média das idades. Assim como em matemática essas estruturas são chamadas de variáveis e servem para guardar e manipular informações em um programa de computador [Quo; Sch]. Porém, na computação, elas podem guardar mais que apenas números.

Tipos de dados em C/C++ podem ser basicamente divididos em numéricos, caracteres e lógicos, como já foi citado na Parte I desta apostila[1]. Os tipos numéricos são **inteiro (int)** e **ponto flutuante (float)**. Os **lógicos (bool)** e **caracteres (char)** [Poi]. Essas palavras chaves vem dos termos em inglês *Integer*, *Floating Point*, *Boolean* e *Characters*, respectivamente.

Cada tipo de dado foi desenvolvido para armazenar dados diferentes.

- **int**: armazena número no conjunto dos inteiros, o que é útil quando é preciso armazenar dados contáveis como números de pessoas em uma sala ou idades.
- **float**: é utilizado para representar números com casa decimais, o que torna conveniente para armazenar a temperatura de ambientes, alturas ou resultados de equações.
- **bool**: assume apenas dois valores verdadeiro ou falso, sendo adequado para guardar informações sobre pressionamento de botão ou se alguém se enquadra ou não em uma categoria.
- **char**: consegue armazenar um caractere como a letra 'a' utilizando aspas simples.

Para criar uma variável é preciso dizer o tipo dela seguido pelo nome da variável. O Código presente em 1.1 mostra como é possível criar uma variável dos 4 tipos descritos anteriormente em C++. Cada tipo de variável ocupa uma quantidade diferente de espaço na memória, o que varia normalmente de 1 a 4 bytes [Poi].

cod:
cria_variavel

```
int main(){
    int variavelInt
    float variavelFloat
    bool variavelBooleana
    char variavelChar
}
```

Listing 1.1: Exemplo básico de como criar uma variável em C/C++.

Entendendo o que foi dito acima, podemos rapidamente entender que para ver o tamanho de uma variável na sua plataforma, o Código 1.2 pode ser utilizado. Portanto vejamos então como funcionaria para sabermos esses espaços na prática.

1. Tipos de dados básicos em C++

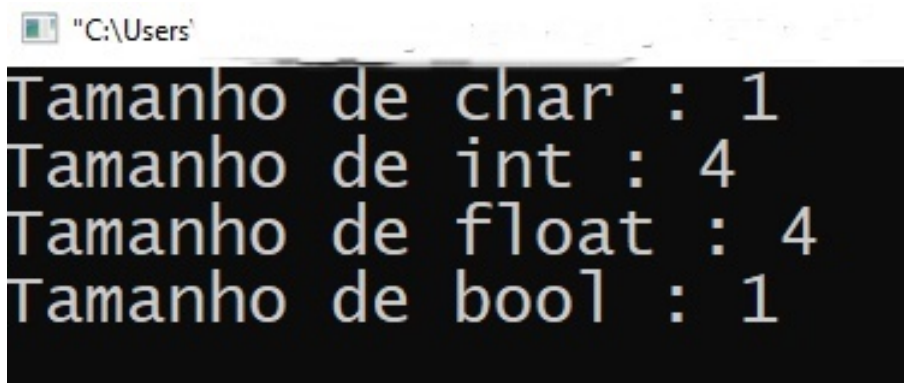
cod:
tamanho_variavel

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

void main() {
    cout << "Tamanho de char : " << sizeof(char) << endl;
    cout << "Tamanho de int : " << sizeof(int) << endl;
    cout << "Tamanho de float : " << sizeof(float) << endl;
    cout << "Tamanho de bool : " << sizeof(bool) << endl;
}
```

Listing 1.2: Exemplo de como imprimir na saída padrão o tamanho de diferentes tipos de variáveis em C++.

O algoritmo acima terá a seguinte saída:



```
"C:\Users'
Tamanho de char : 1
Tamanho de int : 4
Tamanho de float : 4
Tamanho de bool : 1
```

Figura 1.1: Prompt com saída do programa exposto no exemplo 1.2

Com isso, vemos que foi impresso na tela o valor que cada uma das variáveis ocupa na memória.

Além dos tipos de variáveis básicos, existem alguns outros tipos que vale a pena você conhecer, que é o caso do `long`, `short`, `signed`, `unsigned` e `void`, por exemplo. O conceito e tamanho de cada um em bytes pode ser conferido na tabela 1.1.

Tipo	Descrição	Bytes
<i>int</i>	Variável que armazena números inteiros, que por default é do tipo signed, num intervalo de -32.768 a 32.767 , caso seja 2 bytes, ou de $-2.147.483.648$ a $2.147.483.647$, caso ocupe 4 bytes da memória.	2 a 4 bytes
<i>float</i>	Essa variável armazena números reais em um intervalo de $1,2 * 10^{-38}$ a $3,4 * 10^{38}$, com um máximo de 6 casas deprecisão. Por default ela também é signed.	4 bytes
<i>double</i>	Essa variável é análoga ao float, entretanto o intervalo de números que ela armazena vai de $2,3 * 10^{-308}$ a $1,7 * 10^{308}$, tendo uma precisão de até 15 casas decimais.	8 bytes
<i>char</i>	A variável char armazena caracteres, que por sua vez fazem referência à números inteiros, de acordo com a tabela ASCII. Como a variável é por default do tipo signed, ela varia de -128 a 127 .	1 byte
<i>long ou long int</i>	Essa variável é do tipo inteiro e por default ela é signed, bem como uma variável do tipo int. A sua principal diferença de uma variável do tipo int é o tamanho em bytes e o fato de ela poder armazenar números no intervalo de $-9,22 * 10^{-16}$ para $9,22 * 10^{16}$.	8 bytes
<i>long double</i>	É uma variável double só que com mais bytes de armazenamento, portanto, o intervalo que pode ser armazenado nesse tipo de variável é de $3,4 * 10^{-4932}$ a $1,1 * 10^{4932}$, com uma precisão máxima de 19 casas decimais.	10 bytes
<i>short ou short int</i>	A variável do tipo short armazena números do tipo inteiro dentro do intervalo de -32.768 a 32.767 , sendo, portanto, por default do tipo signed.	2 bytes
<i>signed</i>	Toda variável acompanhada pelo signed - como signed int, signed char ou signed float - estará acompanhado de um sinal (+ ou -) de positivo ou negativo. Por default, vários tipos de variáveis são signed, como pode ser visto nessa tabela.	Depende do tipo da variável relacionada (int, float, char, etc)
<i>unsigned</i>	De forma análoga à descrição de signed, uma variável unsigned não guardará consigo o sinal de positivo ou negativo do número armazenado, assim, o seu intervalo irá variar de zero até um valor. No caso do unsigned char, por exemplo, irá variar de 0 a 255.	Depende do tipo da variável relacionada (int, float, char, etc)
<i>void</i>	Esse tipo de variável é utilizado muito em funções, pois ele indica que a variável não poderá armazenar nada. Quando colocado no parâmetro de uma função, estamos dizendo que aquela função não possui parâmetros. Quando dizemos que o retorno de uma função é do tipo void, estamos dizendo que a mesma não possui retorno e portanto, é um procedimento	0 byte

Tabela 1.1: Tipos de dados em C/C++, descrição e tamanho de cada tipo de variável.

tab:tiposVar

1. Tipos de dados básicos em C++

1.1 Conversão de dados em C/C++

É possível que durante a criação de um código você se depare com uma situação em que precise transformar um dado informado pelo usuário, por exemplo, do tipo `int` para um dado do tipo `float`. É exatamente para essas ocasiões que saber converter dados em C/C++ pode ser muito útil.

Conversão de dados em C

Como a linguagem C é um pouco mais antiga que o C++, temos um pouco de dificuldade ao tentarmos converter dados. O que precisamos deixar claro é que existem dois tipos de conversão de dados em C: a implícita e a explícita. Na conversão implícita, não é necessário nenhum operador para conversão. Eles são transformados automaticamente quando um valor é copiado para um outro tipo compatível de dado no programa. Um exemplo disso é a conversão de um número inteiro para um `double`, como podemos ver no exemplo a seguir.

cod:CconvImp

```
#include <stdio.h>

int main(){

    int i = 5;
    double d;

    d = i;

    printf("transformacao implicita: %f \n", d);

    return 0;
}
```

Listing 1.3: Exemplo da conversão de dados implícita em C.

Nesse código temos a saída "transformacao implicita: 5.000000".

Existem duas formas de fazermos a conversão explícita de dados em C, a primeira seria utilizando algumas funções presentes na biblioteca *stdlib.h*, essas funções podem ser cheçadas na tabela 1.2.

Funções para Conversão de dados	Descrição
<i>atof()</i>	Converte string para float.
<i>atoi()</i>	Converte string para int.
<i>atol()</i>	Converte string para long.
<i>itoa()</i>	Converte int para string.
<i>ltoa()</i>	Converte long para string.

Tabela 1.2: Funções nativas do C para conversão de dados específicos.

tab:funcConv

E a segunda forma é adicionando o operador (*tipo*), tal que se quisermos transformar um dado do tipo inteiro para um dado do tipo `short` podemos fazer da seguinte maneira [pro]:

cod:CconvExp

```
#include <stdio.h>

int main(){

    int i = 20;
    short s;

    s = (short) i;

    printf("transformacao explicita: %i \n", s);

    return 0;
}
```

Listing 1.4: Exemplo da conversão de dados explícita em C.

Conversão de dados em C++

Assim como em C, é possível realizarmos conversões de tipos de dados de forma implícita e explícita. A principal diferença entre as conversões de dados em C e em C++ aparece na utilização de *streams* para realizar as conversões de *string* para *int*, *float*, *double*, entre outros, bem como a recíproca disso. Antes de tudo é importante saber o que são *streams*. Uma *stream* é uma abstração que representa um dispositivo no qual as operações de saída e entrada são realizadas[cpl]. Uma *stream* basicamente pode ser reinterpretada como um recurso (*ostream*) ou destino (*istream*) de caracteres com comprimento ou tamanho indefinido.

Para resumir e facilitar o entendimento, podemos dizer que uma *stream* é como um canal que pode receber ou entregar uma sequência de caracteres. Dessa forma, podemos, por exemplo, pedir que um número inteiro armazenado em uma variável chamada *X* fosse impressa na tela do usuário utilizando a *stream* *cout*, tal que sua função é transformar o número inteiro numa sequência de caracteres, por meio do operador «, e só depois imprimí-las na tela. Porém esse é o assunto do nosso próximo capítulo. O que queremos saber aqui é como fazer essas conversões, e para isso iremos precisar da biblioteca *sstream*, que basicamente adiciona *streams* de *strings*, tornando possíveis as conversões citadas anteriormente. De fato podemos perceber que o nome da biblioteca se refere ao termo em inglês *String Stream*. Vejamos um exemplo.

cod:CppStream1

```
#include <iostream> //inclui a biblioteca padrao do C++
#include <sstream> //inclui as string streams
#include <string> //inclui o tipo string (sequencia de caracteres)

using namespace std;

int main(){

    string str = "Pedro tem ";
    int idade = 20;
    stringstream sstream;

    sstream << idade << " anos!"; //armazena a idade de Pedro fazendo a conversao
                                //para o tipo stringstream

    str += sstream.str(); //Concatena o que tem armazenado em str com o que tem
                        //armazenado em sstream

    cout << str << endl; //imprime a variavel str

    return 0;
}
```

Listing 1.5: Exemplo de conversão de dados utilizando streams de strings em C++..

No exemplo acima, utilizamos o método *.str()* para extrair os dados armazenados em *sstream* como dados do tipo *string*. Assim, veremos que a saída do programa será *"Pedro tem 20 anos!"*.

Da mesma forma, podemos também transformar a idade de Pedro de *string* para um número inteiro, como ilustra o exemplo a seguir.

cod:CppStream2

```
#include <iostream> //inclui a biblioteca padrao do C++
#include <sstream> //inclui as string streams
#include <string> //inclui o tipo string (sequencia de caracteres)

using namespace std;

int main(){

    string str = "20";
    int idade;
    stringstream sstream;

    sstream << str; //armazena a idade de Pedro fazendo a conversao
                  //para o tipo stringstream
```

1. Tipos de dados básicos em C++

```
sstream >> idade;      //realiza a conversao do dado stringstream
                        //para inteiro e armazena na variavel idade

cout << idade << endl;  //imprime a variavel idade

return 0;
}
```

Listing 1.6: Exemplo de conversão de dados utilizando streams de strings em C++..

No caso acima, teremos a seguinte saída para o programa: "20", que corresponde justamente à idade de Pedro.

Agora que sabemos como declarar uma variável, a finalidade de cada uma e como convertê-las, podemos seguir para o próximo capítulo, onde iremos aprender um pouco mais sobre entradas e saídas de dados em C/C++.

1.2 Entrada e Saída padrão de dados

Imaginemos o seguinte caso: você precisa criar um programa que recebe dois números inteiros e retorne na tela a soma deles. Neste caso, a entrada de dados será pelo teclado inserindo os dois números e a saída será apresentada em seguida na tela.

Entrada de dados

A entrada de dados que utilizaremos neste começo é a que o usuário insere o argumento a partir do teclado. A regra básica para o bom funcionamento do programa é que o usuário insira o tipo de dado que o programa pede. Exemplo: se pede número **inteiro**, não é sugerível inserir um número **real**. As funções de entrada de dados armazenam o valor que receberam do usuário em uma variável especificada pelo programador.

Na linguagem C, a entrada de argumentos que será apresentada é a função **scanf** [Tre]. Esta função é muito prática para entrada de dados pois especificamos qual tipo de dado ela receberá e em qual ordem deverá ser feita. Para utilizá-la, é preciso colocar o tipo de dado que ela receberá e a variável que irá recebê-la.

Para o argumento "tipo de dado", eles devem ser iniciado por (**porcentagem**) e seguidos de uma letra, dependendo de seu tipo. Iremos trabalhar com os seguintes: **d** ou **i** para números inteiros, **f** para números reais (pontos flutuantes) e **s** para sequências de caracteres (palavras e textos).

Já o argumento variável deverá conter qual variável receberá o valor.

```
#include <stdio.h>

int main() {
    int numDec;
    scanf("%d", numDec);
    return 0;
}
```

Listing 1.7: Exemplo básico de entrada de uma variável em C.

Isso irá colocar na variável **numDec** o valor que o usuário colocar no prompt de comando. É possível receber mais de um tipo de dado em um mesmo scanf, basta acrescentar mais pares de tipo de dado e variável.

```
#include <stdio.h>

int main() {
    int numDec;
    int numDec2;
    float numReal;
    char frase[100];

    scanf("%d %d %f %s", numDec, numDec2, numReal, frase);

    return 0;
}
```

Listing 1.8: Exemplo básico de entrada de duas variáveis em C.

A linguagem C++ segue o mesmo padrão de comportamento do C, podendo até mesmo usar a função scanf. Porém existe um método mais simples e utilizado para entrada de dados em C++, que é o objeto **cin** [Pinb]. A chamada dela é mais simples que a do scanf pois não é preciso dizer qual é o tipo de dado que vai entrar, ela irá subentender a partir do tipo de variável que recebe o dado. Só é necessário acrescentar mais uma linha de código antes de iniciar o principal: using namespace std; (isso nos poupará dores de cabeça nesse começo) 1.9.

1. Tipos de dados básicos em C++

cod:entrada1_cpp

```
#include <iostream>

using namespace std;

int main() {
    int numDec;

    cin >> numDec;
    return 0;
}
```

Listing 1.9: Exemplo básico de entrada de uma variável em C++.

E a chamada para duas variáveis segue o mesmo estilo, basta acrescentar mais operadores >> e chamar a outra variável.

cod:entrada2_cpp

```
#include <iostream>

using namespace std;

int main() {

    int numDec;
    int numDec2;
    float numReal;
    char frase[100];

    cin >> numDec >> numDec2 >> numReal >> frase;

    return 0;
}
```

Listing 1.10: Exemplo básico de entrada de duas variáveis em C++.

Saída de dados

Saída de dados consiste em mostrar algo ao usuário. Isso pode ser o resultado final de uma operação, uma frase escolhida pelo programador ou algo relacionado ao que foi inserido na entrada. Em C, a saída de dados será feita pela função **printf()** [Tre]. Tudo que estiver dentro dos parêntesis irá ser mostrado na tela final do usuário 1.11.

cod:saida1_c

```
#include <stdio.h>

int main() {

    printf("Ola usuario");
    return 0;
}
```

Listing 1.11: Exemplo básico de saída de dados em C.

Para dar a saída em variáveis, é só seguir o exemplo abaixo.

cod:saida2_c

```
#include <stdio.h>

int main() {

    int numDec = 12;
    float numReal = 1.5;

    printf("variaveis receberam %d, %f", numDec, numReal);
    return 0;
}
```

Listing 1.12: Exemplo básico de saída envolvendo variáveis em C.

Em C++ também existe uma simplificação da função `printf()`: o objeto **cout** [Pinb]. O uso do `cout` é mais simples pois, tal qual o `cin`, não precisa declarar o tipo de variável de saída, basta usar os operadores `<<` para dar a saída das variáveis e texto.

cod:saida_cpp

```
#include <iostream>

using namespace std;

int main() {

    int numDec = 12;
    float numReal = 2.3;

    cout<<"variaveis receberam : "<<numDec<<" "<<numReal<<endl;
    return 0;
}
```

Listing 1.13: Exemplo básico de saída envolvendo variáveis em C++.

CAPÍTULO 2

Vetores (Arrays)

sec:arr

2.1 Vetores Unidimensional

Agora suponha que o **número de variáveis** que você quer armazenar é **muito grande ou você ainda não sabe o tamanho**. Por exemplo, suponha que você quer reorganizar uma lista de 10 inteiros de trás para frente.

Uma forma de fazer isso poderia ser:

cod:entrada1_c

```
#include <stdio.h>

int main(){
    // declarando 10 variaveis
    int a1, a2, a3, a4, a5, a6, a7, a8, a9, a10;

    // fazendo a leitura de 10 inteiros
    scanf("%d%d%d%d", &a1, &a2, &a3, &a4, &a5);
    scanf("%d%d%d%d", &a6, &a7, &a8, &a9, &a10);

    // exibindo os 10 inteiros na ordem reversa
    printf("%d %d %d %d %d ", a10, a9, a8, a7, a6);
    printf("%d %d %d %d %d\n", a5, a4, a3, a2, a1);

    return 0;
}
```

Listing 2.1: Invertendo 10 inteiros.

Note que a quantidade de variáveis usadas para esse exemplo foi **bem grande**. Imagine se fossem **100, 1000** ou **um milhão de valores!** Ou se fosse uma **quantidade variável**, por exemplo, caso fossem as notas de alunos de diferentes turmas, cada uma com um número diferente de alunos.

Uma forma bem melhor de fazer isso é utilizar *vetores* (ou *arrays*). Com eles, podemos fazer de maneira similar ao código do exemplo: usar números para identificar variáveis. Esses números são chamados de índices.

Com o uso dos vetores, não é necessário declarar cada variável a ser utilizada. Apenas uma declaração é feita.

Um vetor é basicamente **uma estrutura de dados que permite definir uma sequência de variáveis**, ao invés de apenas uma. Quando declaramos um vetor, precisamos dizer a quantidade de elementos nele, e fazemos isso com o uso dos colchetes:

cod:entrada2_c

```
#include <stdio.h>

int main(){
    int a[10];
    return 0;
}
```

Listing 2.2: Declarando um vetor com 10 inteiros.

2. Vetores (Arrays)

No código acima, observe que a variável *a* não é um inteiro, mas sim um vetor. Em C/C++, o primeiro elemento do vetor possui índice 0. Dessa forma, apesar de “*a*” não ser um inteiro, temos que *a*[0], *a*[1], ..., *a*[9] são sim variáveis de inteiros!

Isso significa que podemos escrever, por exemplo:

cod:entrada3_c

```
#include <stdio.h>

int main(){

    // declarando um vetor com 5 elementos
    int a[5];

    // lendo um de seus elementos
    printf("Digite um valor para o primeiro elemento do vetor: ");
    scanf("%d", &a[0]);

    // atribuindo um valor a outro elemento
    a[1] = 7;

    // exibindo o primeiro e o segundo elemento do vetor
    printf("a[0] = %d, a[1] = %d\n", a[0], a[1]);

    // exibindo a soma desses 2 elementos
    printf("soma = %d\n", a[0] + a[1]);

    return 0;
}
```

Listing 2.3: Manipulando um vetor com 10 inteiros em C.

Por fim, note que podemos usar um **comando de repetição** para realizar a **mesma tarefa com índices diferentes**. Por exemplo, usar o comando *for* para iterar o índice, de 1 até 10, para ler cada elemento do vetor, e depois para imprimir na ordem reversa esses elementos:

cod:entrada4_c

```
#include <stdio.h>

int main(){

    // declarando uma variavel para o tamanho do vetor
    int n = 10;

    //declarando o vetor com N elementos
    int a[n];

    // lendo os elementos do vetor, do primeiro ao ultimo
    for(int i = 0; i < n; i++){

        // lendo o elemento "i" (no inicio i=0, depois i=1, ..., ate i=n-1)
        scanf("%d", &a[i]);
    }

    // imprimindo os elementos do vetor na ordem oposta
    for(int i = n-1; i >= 0; i--){

        // exibindo o elemento "i" (no inicio i=n-1, depois i=n-2, ..., ate i=0)
        printf("%d ", a[i]);
    }
    printf("\n");
    return 0;
}
```

Listing 2.4: Invertendo um vetor.

Note que, dessa forma, é possível ler vários elementos sem ter que declarar e usar uma quantidade imensa de variáveis. Se quiséssemos, ao invés de 10, ler 100 elementos, bastaria alterar a atribuição da variável *n* para *n* = 10 no início do código!

2.2 Vetores com mais de uma dimensão (Matrizes)

Seguindo o mesmo raciocínio, podemos armazenar variáveis em forma de **matrizes**. Uma **matriz** nada mais é que uma estrutura de dados do tipo vetor que pode ter duas ou mais dimensões.

Assim como os vetores unidimensional, as matrizes irão armazenar dados de um mesmo tipo (*int*, *float*, *char*, etc). Na prática, uma matriz aloca uma tabela na memória.

Saindo um pouco da teoria, vamos aprender como declarar uma matriz **na prática**:

cod:entrada5_c

```
#include <stdio.h>

int main(){

    //declarando uma matriz 3x2 de numeros inteiros
    int matriz[3][2];

    return 0;
}
```

Listing 2.5: Declarando uma matriz 3x2.

Perceba que ao declararmos a matriz `matriz[3][2]` estamos dizendo para o nosso computador que queremos que ele guarde uma espaço na memória em formato de uma tabela que tera *3 linhas* e *2 colunas*. Podemos criar uma matriz com a quantidade de linha ou colunas que quisermos. No exemplo abaixo, iremos criar uma matriz para armazenar os nomes (*strings*) de algumas pessoas. Para isso, perceba que temos que definir o tamanho das *strings* (número de caracteres de cada nome) e a quantidade de nomes que vamos querer.

cod:entrada6_c

```
#include <stdio.h>

int main(){

    //declarando uma matriz 3x7 de caracteres que armazena strings
    char matriz[3][7] = {"Lucas", "Felipe", "Rennan"};

    return 0;
}
```

Listing 2.6: Declarando uma matriz com string.

Assim, a nossa próxima etapa é saber como imprimir e alterar um ou mais termos de uma matriz.

Vamos pensar na matriz como sendo um campo minado. Quando jogamos campo minado temos que dizer um número/letra que representa a linha e outro que representa a coluna, dessa forma, o nosso adversário vai saber onde está sendo a nossa jogada. Em **C/C++** quando falamos de **matrizes** o pensamento é muito parecido. Suponha que você quer atribuir valores para uma matriz 3x3, logo em seguida você quer mudar o primeiro valor para 0 e os demais por valores digitados pelo usuário, depois disso, você quer imprimir cada item dessa matriz, como você faria isso?

Bem, existem duas formas de fazer, como você já viu nos *arrays* de uma dimensão, porém iremos mostrar apenas o jeito mais prático, a forma mais usual, que é utilizando dois laços *for*, onde a variável utilizada no primeiro laço percorrerá os valores das linhas da matriz, retornando assim as suas colunas, enquanto a variável do segundo laço percorrerá os valores das colunas, retornando assim seus elementos. Agora, veja o exemplo de como ficaria esse código.

cod:entrada7_c

```
#include <stdio.h>

int main(){

    //declarando uma matriz 3x3 de numeros inteiros e atribuindo numeros aleatorios
    int matriz[3][3] = {{1,2,3},{4,5,6},{7,8,9}};

    //mudando o valor do primeiro elemento da matriz[3][3] para 0
}
```

2. Vetores (Arrays)

```
matriz[0][0] = 0;

//estrutura que percorre as m linhas da matriz
for(int m = 0; m < 3; m++)
{
    //estrutura que percorre as n colunas da matriz
    for(int n = 0; n < 3; n++)
    {
        /*imprime uma mensagem para o usuario informando qual eh o
        elemento que ira receber o valor do usuario*/
        printf("Informe o valor do Elemento[%i][%i]: ", m, n);
        /*armazena o valor informado pelo usuario no elemento
        matriz[m][n]*/
        scanf("%i", &matriz[m][n]);
    }
}

//estrutura que percorre as m linhas da matriz
for(int m = 0; m < 3; m++)
{
    //estrutura que percorre as n colunas da matriz
    for(int n = 0; n < 3; n++)
    {
        /*imprime uma mensagem para o usuario informando qual o
        valor de cada elemento da matriz*/
        printf("Elemento[%i][%i]: %i\n", m, n, matriz[m][n]);
    }
}

return 0;
}
```

Listing 2.7: Manipulação de matrizes.

O termo $\&matriz[m][n]$ retorna o **endereço de memória** do elemento indexado à linha m e coluna n da matriz de números inteiro, enquanto $matriz[m][n]$ retorna o **valor** do elemento indexado à linha m e coluna n da mesma matriz.

Acredito que até aqui o conceito de vetores e matrizes já se mostram bem mais claros, entretanto, vale dizer que um array pode ter n -dimensões, tal que a dificuldade de lidar com eles vai aumentando de acordo como número de dimensões que escolhemos dar para eles, por isso restringir o nosso mundo a no máximo três dimensões é uma boa prática de programação quando estamos trabalhando com arrays.

Nesse capítulo você aprendeu a estrutura de dados mais utilizada por todos os programadores ao redor do mundo, os Arrays. No próximo capítulo iremos relembrar o que são Estruturas de Decisão e como utilizá-las em um código de C/C++.

CAPÍTULO 3

Estruturas de decisão

sec:dec

3.1 Definição e condições básicas

Ao tratarmos de estruturas de decisão em C++, temos que ter em mente que o mesmo se definirá pelo fato de que vão existir condições e limitações definidas pelo usuário a respeito do que o programa irá ou não executar. No C++, veremos que é possível se utilizar de diversas condições matemáticas básicas e de uso casual. Por exemplo:

Tabela 3.1: Condições básicas na matemática e suas respectivas equivalência em C++.

Condições matemáticas básicas	Como isso é em C++?
<i>Menor que: $a < b$</i>	$a < b$
<i>Menor ou igual a: $a \leq b$</i>	$a \leq b$
<i>Maior que: $a > b$</i>	$a > b$
<i>Maior ou igual a: $a \geq b$</i>	$a \geq b$
<i>Igual a: $a = b$</i>	$a == b$
<i>Diferente de: $a \neq b$</i>	$a != b$

Sabendo disso, vamos ao próximo passo!

3.2 Comando If-Else

Agora que entendemos algumas condições matemáticas e suas aplicações em C++, podemos então começar a falar dos comandos de decisão, no caso o **Switch/case** e o **If-else**, contudo, vamos por ora nos prender ao comando if-else. Mas o que é esse comando? É uma estrutura condicional da linguagem a qual estamos estudando, que vai receber uma condição e daí, portanto, definir se essa condição é verdadeira ou falsa e logo após executar o comando que é pedido baseado na condição proposta. Vamos simplificar:

cod:estrutura

```
if ( condicao ) {  
    //comando que deve ser executado caso a condicao seja verdadeira.  
} else  
    //comando que deve ser executado caso a condicao seja falsa.
```

Listing 3.1: Exemplo básico da estrutura do comando .

O que queremos dizer com essa estrutura? Ora, de uma maneira bruta podemos dizer que seria: if [SE] (se a Condição é verdadeira) execute o comando abaixo, else [SENÃO] execute o próximo comando. Vejamos um exemplo simples na prática:

O comando abaixo pede pra entrarmos com dois números inteiros e eles nos dirá se os números digitados são iguais, ou no caso de algum de algum deles ser maior, o programa vai nos dizer qual é!

3. Estruturas de decisão

cod:exemplo

```
#include <iostream>

using namespace std;

int main (void)
{
    int N1, N2 ;
    cout << ( "Digite o primeiro numero: " ) << endl;
    cin >> (N1);
    cout << ( "Digite o segundo numero: " ) << endl;
    cin >> (N2);

    if (N1 == N2){
        cout << ( "Os numeros sao iguais!" ) << endl;
    }
    else{
        if (N1 > N2){
            cout << "O maior valor e =" << N1 ;
        }
        else{
            cout << "O maior valor e =" << N2;
        }
    }

    return 0;
}
```

Listing 3.2: Exemplo básico de um código if-else em C++.

Para que você não saia dessa apostila sem sequer saber como é a estrutura do Switch/Case, deixaremos um exemplo básico dessa estrutura de decisão.

cod:Switch/Case

```
#include <stdio.h>

int main (void)
{
    int numQualquer = 0;
    printf("Informe um numero inteiro de 0 a 5: ");
    scanf("%d", &numQualquer);
    print("\n"); //pula uma linha

    switch(numQualquer){
        case 0: //testa se numQualquer == 0
            printf("O numero informado foi o zero. \n");
            break;
        case 1: //testa se numQualquer == 1
            printf("O numero informado foi o um. \n");
            break;
        case 2: //testa se numQualquer == 2
            printf("O numero informado foi o dois. \n");
            break;
        case 3: //testa se numQualquer == 3
            printf("O numero informado foi o tres. \n");
            break;
        case 4: //testa se numQualquer == 4
            printf("O numero informado foi o quatro. \n");
            break;
        case 5: //testa se numQualquer == 5
            printf("O numero informado foi o cinco. \n");
            break;
        default: //se nenhuma das condicoes forem aceitas, retorna default
            printf("O numero informado nao eh valido.");
            break;
    }

    return 0; //Encerra a funcao main() retornando zero (programa executou certo)
}
```

Listing 3.3: Exemplo básico de um código utilizando a estrutura de decisão Switch/Case em C.

Neste pequeno capítulo, pudemos ter uma ideia geral do if-else [w3s] e também vimos a existência do switch/case. No próximo capítulo iremos expandir o nosso mundo na programação um pouco mais, explorando todas as estruturas de repetição que irão nos ajudar bastante a simplificar códigos em looping.

CAPÍTULO 4

Estruturas de repetição

sec:rep

Comumente, os programadores se deparam com situações em que um mesmo comando precisa ser realizado várias vezes, logo você pode pensar: "basta então eu fazer um só código e copiar ele o número de vezes que eu precisar!" vamos ver se isso é viável? Suponhamos que pretendemos criar um algoritmo que incrementa uma variável do tipo inteiro de valor inicial igual a zero até que ele atinja um determinado valor, por exemplo dois, e que imprima os valores que a variável assume a cada incremento.

cod:repeticao1_c

```
#include <iostream>

using namespace std;

int i=0;

int main(){
    i++;
    cout<<i << " ";
    i++;
    cout<<i;
}
```

Listing 4.1: Exemplo básico de incremento de uma variável.

Apesar de funcionar para esse caso específico, o Exemplo 4.1 apresenta uma má técnica ou vício de programação, pois utiliza uma mesma instrução mais de uma vez.

Temos que ter em mente o fato de que poderíamos ser solicitados a solucionar o mesmo problema com a diferença de que, em vez de 2, precisássemos atingir o valor 1000.

Evidentemente, não seria conveniente escrever um mesmo comando mil vezes. Para lidar com problemas desta ordem, existem as estruturas de repetição, que permitem que realizemos uma ação várias vezes sem precisarmos declarar a instrução para tal ação todas as vezes em que ela deve ser realizada.

As duas estruturas de repetição que apresentaremos são while e o for.

4.1 Comando while

O comando while permite que um certo trecho de programa seja executado ENQUANTO uma certa condição for verdadeira[Pina][San]. A forma do comando é a seguinte:

cod:repeticao2_c

```
while (condicao)
{
    // comandos a serem repetidos
    // comandos a serem repetidos
}
// comandos apos o 'while'
```

Listing 4.2: Forma do comando while.

4. Estruturas de repetição

O `while` testa, inicialmente, uma condição dada pelo programador. Caso a condição seja verdadeira, os comandos contidos no bloco do `while` (entre chaves, como vemos no Exemplo 4.2). Ao executar o último comando dentro do bloco, o compilador testa novamente a condição. Se a condição ainda for verdadeira, o processo se repete. Caso contrário, será executado o que estiver fora do laço, isto é, após essa estrutura.

O Exemplo 21 mostra uma solução para o problema proposto anteriormente (incrementar a variável inteira até que ela atinja o valor 1000) com o uso do comando `while`.

cod:repeticao3_c

```
#include<iostream>

using namespace std;

int i=0;

int main(){

    while (i<1000){
        i++;
        cout<<i<< " ";
    }
}
```

Listing 4.3: Exemplo básico de incremento de uma variável utilizando a estrutura de repetição `while`.

4.2 Comando `for`

O comando `for` permite que um certo trecho de programa seja executado um determinado número de vezes [Pina][San]. Segue a seguinte estrutura:

cod:repeticao4_c

```
for (comandos de inicializacao;condicao de teste;incremento/decremento)
{
    // comandos a serem repetidos
    // comandos a serem repetidos
}
// comandos apos o 'for'
```

Listing 4.4: Forma do comando `for`.

De modo análogo ao que ocorre no `while`, os comandos contidos no bloco são repetidos enquanto a condição determinada for verdadeira. Seus parâmetros são:

- **Variável de inicialização:** comando de atribuição que inicia uma variável que faz o controle do laço.
- **Condição:** expressão usada que determinará o final do laço.
- **Incremento:** define a variável de controle e muda a cada passada no laço.

A variável de controle sofre uma alteração a cada ciclo. Com base nela, a condição declarada na estrutura `for` determinará a "parada" do laço.

O Exemplo 4.3 mostra um algoritmo que atribui aos elementos de um vetor de cinco unidades valores definidos pelo usuário via teclado e os imprime em seguida.

cod:repeticao5_c

```
#include<iostream>

using namespace std;
```

```
int i=0;
int vetor [5];

int main() {

    for (i=0; i<5; i++){
        cin>>vetor[i];
    }

    for (i=0; i<5; i++){
        cout<<vetor[i]<<" ";
    }

}
```

Listing 4.5: Exemplo básico de incremento de uma variável.

Este capítulo foi dedicado às estruturas de repetição. Nele você conseguiu aprender as principais funções do C e C++ utilizadas mundialmente para a criação de *loopings* que podem ser úteis e estritamente necessários quando bem utilizados, mas também podem causar uma dor de cabeça caso o mesmo não se encerre, como é o caso do *looping* infinito comumente ocasionado pela má utilização das funções *while* e *do-while*. O importante é que agora você está apto a passar para o próximo capítulo, cujo assunto compreende os principais conceitos sobre **Funções**, o que são, para que servem e como utilizá-las de forma correta. Nos vemos no próximo capítulo!

CAPÍTULO 5

Funções

sec:func

Uma função nada mais é do que uma subrotina usada em um programa. Uma espécie de receita de bolo que deve ser repetida todas as vezes que precisarmos dela [Cas].

Em C/C++, uma função consiste no conjunto de comandos que realizam uma função específica. Além disso, toda função em C/C++ retornará um tipo de variável específica, como *int*, *float*, *boolean*, entre outros. É exatamente por isso que toda função em C/C++ deve ser definida antes de ser implementada, analogamente à declaração de uma variável, como mostra o exemplo abaixo.

cod:func1c

```
#include <stdio.h>

int soma (int a, int b){
    return a + b;
}
```

Listing 5.1: Exemplo de uma função C.

A função acima (Exemplo 5.1) tem o objetivo de somar duas variáveis do tipo inteiro *a* e *b*. Dessa forma, note que o resultado dessa função também será do tipo inteiro, justamente por isso declaramos a função *soma* como sendo do tipo inteiro.

Toda função em C irá seguir a seguinte estrutura:

cod:func2c

```
tipo nome_da_funcao (tipo <parametro_1>, tipo <parametro_2>, ..., tipo <parametro_n>){
    instrucoes;
    return (valor_de_retorno);
}
```

Listing 5.2: Esquema de uma função genérica em C.

Os parâmetros citados anteriormente no exemplo 5.1 e 5.2 são as variáveis declaradas diretamente no cabeçalho da função. A finalidade desses parâmetros é ligar as funções auxiliares que estamos criando com as função principal. As passagens de valores entre uma função e outra nós chamamos de parâmetros [Cas]. Voltemos ao exemplo 5.1 mas agora comentando o código e com algumas mudanças.

5. Funções

cod:func3c

```
#include <stdio.h>
//Implementando uma funcao que soma dois parametros de uma funcao principal
qualquer.

//Primeiro temos que definir o tipo, o nome e os parametros da funcao, nesta ordem.
int soma (int a, int b){
//"int a" e "int b" sao os dois parametros da funcao "soma"

    //Declarando as variaveis
    int c;

    c = a + b;
    //Perceba que a instrucao eh somar os dois parametros e armazena-los em outra
    variavel "c"

    return c;
    //E o resultado sera o retorno do valor de "c", isto eh, "a + b"
}
```

Listing 5.3: Exemplo detalhado de uma função em C que retorna a soma de dois parâmetros.

Desta forma, sempre que quisermos somar duas variáveis do tipo inteiro, basta chamá-las na nossa função principal e dizer quais são as duas variáveis que queremos somar, seguindo a seguinte chamada *soma(x,y)*, que poderá ser armazenada em uma variável chamada *resultado*, por exemplo, tal que *resultado = soma(x,y)*. Perceba que não precisamos dizer o tipo do parâmetro ou da variável que será utilizada na função, uma vez que ambos já foram declarados na construção da função *soma()* e da função principal *main()*, respectivamente. Vejamos um exemplo de chamada dessa função soma.

cod:func4c

```
#include <stdio.h>

int soma (int a, int b);

int main(void){
    int a = 3;
    int b = 5;
    int resultado = 0;

    resultado = soma(a, b);

    printf("Resultado da soma de %d e %d eh %d", a, b, resultado);
    //Uma outra forma de imprimir o resultado seria,
    printf("Resultado da soma de %d e %d eh %d", a, b, soma(a, b));

    return 0;
}

int soma (int a, int b){
    return a+b;
}
```

Listing 5.4: Exemplo da chamada/utilização da função soma() na função principal main().

O exemplo acima também é útil para explicarmos a forma certa de uma função em C/C++ ser declarada. Toda função auxiliar em C/C++ deve ser declarada **ANTES** da função principal, o que não impede dessa função ser implementada após a implementação da função principal *main()*, como é o caso do exemplo acima 5.5, onde podemos perceber que a função *soma()* foi declarada antes da função *main()* e sua implementação está disposta no código após a implementação da função principal. Entretanto, podemos optar por declarar e implementar a função *soma()* antes da função *main()*, o que não nos causaria nenhum problema.

5.1 Procedimentos

Também é possível criar funções que não possuem nenhum retorno. Esse tipo de função é chamada de procedimento. A finalidade de um procedimento é executar uma sequência de passos lógicos, como por exemplo, podemos fazer um procedimento que imprima alguma coisa, nesse caso usaremos "Hello World". Como não retornaremos nenhum valor e não utilizaremos nenhum parâmetro, podemos usar qualquer tipo de função e retornarmos um dado aleatório adequado para o tipo de escolhido para a função. Entretanto, é uma boa prática colocarmos o tipo void tanto para o parâmetro quanto para a função.

cod:func4c

```
#include <stdio.h>

//Declarando o tipo, nome e o parametro do procedimento
void hello_world(void){

//Instrucao da nossa funcao hello_world
printf("*****\n");
printf("    HELLO WORLD\n");
printf("*****\n");
}

//Funcao principal
int main(){

//Chamando a funcao que criamos
hello_world();

return 0;
}
```

Listing 5.5: Exemplo de um procedimento em C que imprime "Hello World" e alguns asteriscos.

5.2 Arrays como argumento de uma função ou procedimento

Até então, aprendemos como é o escopo de uma função e de um procedimento, bem como as principais diferenças entre elas, porém ainda não vimos como passar arrays n-dimensionais como parâmetro. Para isso, devemos primeiro entender o conceito da passagem de parâmetros por **cópia** e por **referência**.

Passagem de parâmetros por cópia

Imagine que você está querendo implementar um função que irá auxiliar o seu programa, esta função incrementa cinco nos valores armazenados em duas variáveis, que por sua vez foram declaradas na sua função principal `main()`, e retorna a soma desses dois valores. Entretanto, você deseja manter os valores dessas variáveis na função principal inalterados, pois utilizará esses valores posteriormente no seu código, e não seria nada interessante que esses valores estivessem somados com cinco. Para resolver esses problemas, o C/C++ criaram a passagem de parâmetros/valores por cópia. Toda variável em C ou em C++, independente do seu tipo, desde que não seja um array, será passada como cópia, isto é, qualquer alteração que você faça dentro da sua função não modificará os valores das suas variáveis na sua função principal. Vejamos um exemplo.

cod:funcCcopy

```
#include <stdio.h>

//Declarando o tipo, nome e o parametro da funcao
int soma_sem_modificar(int a, int b){

    a = a + 5;
    b = b + 5;

    int soma = a + b;
}
```

5. Funções

```
        return soma;
    }

//Funcao principal
int main(){

    int a = 5;
    int b = 10;

    //Chamando a funcao que criamos e armazenando o retorno dela numa variavel
    resultado
    int resultado = soma_sem_modificar(a, b);

    printf("%i %i %i \n", resultado, a, b);

    return 0;
}
```

Listing 5.6: Exemplo de uma função em C com passagem por cópia.

A saída do programa acima seria, portanto, 25 5 10.

Passagem de parâmetros por referência

Na passagem de parâmetros por referência, estaremos utilizando uma referência para a variável que estamos passando no parâmetro, tal que toda e qualquer modificação nos valores dessas variáveis que fizermos dentro da função, irá alterar diretamente os seus valores na função principal, sejam elas uma variável com um único dado, como `int x = 5`, ou um array.

Para explorarmos melhor esse tipo de passagem de valores para a função, devemos estudar previamente o que são ponteiros e como utilizá-los. Esse assunto será abordado na segunda parte da nossa apostila, portanto não poderemos falar muito mais sobre ela. Deixaremos um exemplo para aumentar a sua curiosidade!

cod: funcCref

```
#include <stdio.h>

//Declarando o tipo, nome e o parametro da funcao
int soma_modificando(int *pa, int *pb){

    *pa = *pa + 5;
    *pb = *pb + 5;

    int soma = *a + *b;

    return soma;
}

//Funcao principal
int main(){

    int a = 5;
    int b = 10;

    //Chamando a funcao que criamos e armazenando o retorno dela numa variavel
    resultado
    int resultado = soma_sem_modificar(&a, &b);

    printf("%i %i %i \n", resultado, a, b);

    return 0;
}
```

Listing 5.7: Exemplo de uma função em C com passagem por referência..

Passagem de um array como parâmetro de uma função em C/C++

Para passarmos um array como parâmetro de uma função na linguagem C, precisamos ter um conhecimento prévio sobre ponteiros, uma vez que a passagem de um array é feita criando um ponteiro para o array, portanto, não vamos nos aprofundar nessa linguagem. Um exemplo de um procedimento seria:

cod: funcCarr

```
#include <stdio.h>

//Declarando o tipo, nome e o parametro da funcao
int soma_elementos_array(int n, int *vetor){

    int soma = 0;

    for(int i = 0; i < n; i++){
        soma = soma + vetor[i];
    }

    return soma;
}

//Funcao principal
int main(){

    int vnotas[5] = {5, 10, 15, 20, 25};

    //Chamando a funcao que criamos e armazenando o retorno dela numa variavel
    resultado
    int resultado = soma_elementos_array(5, vnotas);

    printf("%i \n", resultado);

    return 0;
}
```

Listing 5.8: Exemplo de uma função em C com um array como parâmetro.

Já em C++, a passagem de um array, seja ele um vetor ou uma matriz, é sempre feita por referência, portanto, não precisamos deixar explícito que o parâmetro é um ponteiro. Existem três formas diferentes de dizer que a variável que estamos utilizando como parâmetro é um array [Tut], entretanto, iremos trabalhar apenas com duas delas - uma vez que a terceira é idêntica - sendo elas a passagem especificando o tamanho do array unidimensional e não especificando, vejamos o exemplo abaixo.

cod: funcCpparr

```
#include <iostream>

using namespace std;

void altera_var_arr(int var, int arr[], int arr_lenght){ //Sem definir o tamanho do array

    var += 5;

    for(int i=0; i < arr_lenght; i++){
        arr[i] *= 2;
    }
}

void altera_var_arr_limitado(int var, int arr[5]){ //Definindo o tamanho do array

    var += 5;

    for(int i=0; i < 5; i++){
        arr[i] *= 2;
    }
}
```

5. Funções

```
int main(void){
    int var = 0;
    int arr[5] = {1, 2, 3, 4, 5};

    cout << "Variavel antes = " << var << endl;
    cout << "Array antes = ";
    for (int i = 0; i < sizeof(arr)/sizeof(arr[0]); i++){
        cout << arr[i] << ' ';
    }
    cout << endl;

    altera_var_arr(var, arr, sizeof(arr)/sizeof(arr[0]));

    cout << "Variavel depois = " << var << endl;
    cout << "Array depois = ";
    for (int i = 0; i < sizeof(arr)/sizeof(arr[0]); i++){
        cout << arr[i] << ' ';
    }
    cout << endl;

    return 0;
}
```

Listing 5.9: Exemplo de uma função em C com um array como parâmetro.

Neste capítulo exploramos o universo das funções e dos procedimentos. É uma boa prática de programação "limparmos" a nossa função principal por meio de abstrações, isto é, transformar blocos de código da função principal em funções auxiliares. Esses blocos de código devem ser escolhidos de acordo com a sua função, por exemplo, se quisermos formar um quadrado na tela do usuário do nosso programa, iríamos gastar cerca de 10 a 20 linhas de código para isso, ao invés de "sujarmos" a nossa função principal com algo que não interfere na **lógica** do nosso programa, podemos simplesmente criar uma função auxiliar chamada *formaQuadrado()* que iria fazer o trabalho pesado para nós. Assim, o nosso código principal ficaria "limpo" e simples, seguindo uma sigla um tanto quanto engraçada chamada de **KISS** que significa *Keep It Simple, Stupid*.

No próximo capítulo iremos estudar a parte de bibliotecas. Artificio que nos ajuda muito na nossa jornada como programador, seja utilizando bibliotecas que já foram criadas por outros programadores para simplificar a nossa vida, seguindo aquele ditado "Não queria reinventar a roda", como também criando as nossas próprias bibliotecas, a fim de manter o nosso programa "limpo" e simples. Veja você no próximo capítulo!

CAPÍTULO 6

Bibliotecas em C/C++

Nesse capítulo vamos expor as principais bibliotecas em C/C++ e suas principais funcionalidades, mas não somente isso. Como foi discutido na primeira parte dessa apostila, é possível que você precise criar suas próprias bibliotecas, portanto, a segunda parte desse capítulo será totalmente dedicado à isso.

6.1 Principais bibliotecas

Da linguagem C

Podemos resumir as principais bibliotecas padrão em C à oito bibliotecas, sendo elas [Feo]:

- ***stdio.h*** - É a biblioteca padrão do C que inclui funções de entrada e saída, como as funções *printf()* e *scanf()*. Você sempre utilizará essa biblioteca em qualquer programa que você fizer.
- ***stdlib.h*** - Essa biblioteca é bastante utilizada quando queremos trabalhar com ponteiros, mas também possui funções muito úteis como as funções de conversão de dados explicados no Capítulo 1 dessa apostila[9]. Além disso, essa biblioteca conta com funções que geram números pseudoaleatórios, como a função *rand()*.
- ***math.h*** - Inclui várias funções e constantes matemáticas. No caso das funções, as principais são as funções de logaritmo, potenciação e as funções trigonométricas.
- ***string.h*** - A utilidade dessa biblioteca é a manipulação de strings. Nela encontramos funções para calcular o tamanho da string, copiar uma string para outra e comparar duas strings.
- ***limits.h*** - Essa biblioteca contém constantes que especificam os tamanhos máximo e mínimo de vários tipos de dados básicos no seu computador.
- ***ctype.h*** - Basicamente essa biblioteca inclui funções que facilita indentificar qual o tipo de caractere(s) que estamos trabalhando, como por exemplo saber se o caractere está maiúsculo ou minúsculo, saber se é um dígito ou uma letra, entre outros.
- ***time.h*** - Como o próprio nome sugere, essa biblioteca nos fornece algumas funções e constantes que nos auxiliam na contagem do tempo, usualmente utilizada para cronometrar programas.
- ***stdbool.h*** - Como em C um valor verdadeiro (*true*) é interpretado como o inteiro 1 e o valor falso (*false*) é interpretado como o inteiro 0, a biblioteca *stdbool* adiciona os termos *false* e *true* como constantes que pode ser atribuídas a qualquer variável do tipo *bool*.

Da linguagem C++

Já em C++, além das bibliotecas padrão do C que foram readequadas para C++, como é o caso das bibliotecas *cctype*, *cmath*, *cstdlib*, *ctime*, *cstdio*, *climits* e *cstring*, temos nove principais bibliotecas bastante utilizadas em C++, sendo elas [Wik]:

- **<iostream>** - É a biblioteca mais básica do C++, análoga a função *stdio.h* do C. Com ela conseguimos manipular fluxo de dados padrão do sistema, como a entrada e saída padrão, e a saída de erros padrão, através dos objetos *cin*, *cout*, *cerr* e *clog*. Bem como adiciona alguns métodos para o controle desse fluxo de dados.
- **<algorithm>** - Fornece diversos algoritmos genéricos úteis para busca, ordenação e transformação de containers (estrutura de dados presente na segunda apostila), entre outros.
- **<fstream>** - Essa biblioteca é utilizada para a manipulação de fluxo de dados de arquivos de computador especializado para o tipo de dado nativo *char*. Dessa forma, ela nos permite ler e escrever, em modo de texto ou binário, os arquivos de um computador.
- **<locale>** - Essa biblioteca manipula diversas convenções culturais do utilizador/usuário, como a representação de números, moedas e datas, para efeitos de internacionalização.
- **<map>** - Adiciona o objeto *map*, que associa um determinado dado (*Data*, o valor) a uma chave (*Key*), criando uma estrutura de dados parecido com um vetor, porém com o índice do objeto sendo a chave que você atribuiu. Falaremos mais sobre essa biblioteca na nossa segunda apostila
- **<set>** - Essa biblioteca adiciona uma nova estrutura de dados chamada *set*, que atribui o valor do elemento como sua própria chave. Por esse motivo, cada valor (e, portanto, sua chave) é único, não pode repetir. Também falaremos mais sobre essa biblioteca na nossa segunda apostila
- **<sstream>** - Adiciona a classe *stringstream*, um manipulador de fluxos de dados de cadeias de caracteres especializado para o tipo de dado nativo *char*. Ele permite ler e escrever, em modo de texto ou binário, *strings*(sequência de caracteres).
- **<string>** - Essa biblioteca adiciona uma cadeia de caracteres especializada para o tipo de dado nativo *char*. Ela remove vários dos problemas introduzidos pela linguagem C ao confiar no programador o gerenciamento de cadeias de caracteres, encapsulando internamente rotinas e considerações que o programador não precisa tomar conhecimento. Ele também permite conversão de e para cadeias de texto do C (*const char**).
- **<vector>** - Essa biblioteca adiciona um novo tipo de vetor mais genérico, que pode ser acessado através de índices para os elementos, assim como em C e sua memória é alocada de forma contígua. Entretanto, a principal diferença para um vetor comum (*array* unidimensional) é o seu tamanho dinâmico - não precisamos definir o tamanho do vetor quando o criamos - com gerenciamento automático e há uma flexibilidade maior para adicionar elementos.

6.2 Como criar e utilizar a sua biblioteca própria

Criar sua própria biblioteca não é difícil, a dificuldade está em como e quando utilizá-la da forma correta, bem como definir as classes, funções e constantes que serão declaradas na sua biblioteca.

Antes de começarmos, é importante saber que toda biblioteca vai ser composta por duas partes, a interface, ou arquivo de cabeçalho, e a implementação. Na interface você estará colocando todas as funções, classes e constantes que você quer utilizar na sua biblioteca, enquanto na implementação você irá colocar, obviamente, a implementação daquelas funções e classes que você criou no cabeçalho.

Assim sendo, temos que para criar e utilizar uma biblioteca devemos seguir cerca de cinco passos [Dep]:

- 1 Criar uma INTERFACE para a sua biblioteca: *mylib.h*
- 2 Criar uma IMPLEMENTAÇÃO da sua biblioteca: *mylib.c*
- 3 Criar um ARQUIVO DE OBJETO BIBLIOTECA (.o) que pode ser ligado com os programas cuja biblioteca fora declarada.

- 3a ou criar um ARQUIVO DE OBJETO COMPARTILHADO (.so) de vários arquivos .o que podem ser ligados dinamicamente com os programas que estão utilizando a sua biblioteca
- 3b ou criar uma FILA DE ARQUIVO (.a) de vários arquivos .o que podem ser estaticamente ligados com programas que usem a sua biblioteca
- 4 USAR a biblioteca em outros códigos em C/C++:
 - a) `#include "mylib.h"`
 - b) *linkar* o código da biblioteca dentro do arquivo *a.out*
- 5 Preparar o variável de ambiente LD_LIBRARY_PATH para achar objetos compartilhados em localizações não-padrão na execução

Abaixo, vamos detalhar cada um dos casos para que fique mais fácil de serem entendidos.

INTERFACE

A *header file* ou arquivo de cabeçalho da sua biblioteca, como já foi citado antes, deve conter de forma explícita todas as funções, classes e constantes que você utilizará na sua biblioteca. Sempre que você criar esse arquivo, o mesmo deve começar com `#ifndef nome_do_arquivo` e terminar com `#endif`, ou, no caso de alguns compiladores mais novos, basta utilizar o `#pragma once` no início do arquivo. Além disso, estaremos utilizando `#define` para definirmos uma constante. Confira o exemplo.

cod:mylib.h

```
//nome do arquivo: mylib.h

#pragma once
#include "stdio.h"

#define PI 3,1415; //define a constante PI

int soma_int(int a, int b); //declara a funcao soma de inteiros

#define MAX_FOO 20; //define a constante MAX_FOO

void ola_mundo(void); //declara a funcao ola mundo
```

Listing 6.1: Exemplo de como criar a interface de uma biblioteca em C/C++ (mylib.h).

É possível adicionar *structs* e classes, mas esses serão nossos objetos de estudo apenas da próxima apostila.

IMPLEMENTAÇÃO

O nome do arquivo da implementação deve ser igual ao da interface, exceto pelo fato de que o arquivo de interface é um arquivo .h e o de implementação deve ser um arquivo .c. A estrutura do arquivo é similar, entretanto, não precisamos utilizar a linha de código `#pragma once`, ao invés disso, vamos incluir a nossa biblioteca através da linha de código `#include "mylib.h"`. Veja o exemplo abaixo.

cod:mylib.c

```
//nome do arquivo: mylib.c

#include "mylib.h" //incluindo a interface da minha biblioteca
#include "stdio.h" //incluindo a biblioteca padrao

//Como ja definimos a constante PI na mylib.c, nao podemos definir novamente

int soma_int(int a, int b){ //implementacao da funcao soma de inteiros
    return a+b;
}

//A mesma logica da constante PI vale para a constante MAX_FOO
```

6. Bibliotecas em C/C++

```
void ola_mundo(void){ //implementacao da funcao ola mundo
    printf("Ola mundo!");
}
```

Listing 6.2: Exemplo de como criar a implementação de uma biblioteca em C/C++ (mylib.c).

Obviamente que as funções e até mesmo as constantes escolhidas não fazem muito sentido e não são nem um pouco úteis, mas a nível de entendimento, espero que tenha ficado tudo claro. Sendo assim, vamos para o próximo passo!

Criar um ARQUIVO DE OBJETO BIBLIOTECA

Para criar um arquivo de objeto biblioteca que pode estar conectado com outros programas que estejam usando a sua biblioteca, basta usar a opção `-c` no seu compilador, para mandar o gcc criar um arquivo objeto (arquivo `.o`) ao invés de um arquivo executável (como é o caso do seu `main.c`). Dessa forma, poderíamos compilar o nosso arquivo com a seguinte linha de código no terminal do nosso compilador:

```
$ gcc -o mylib.o -c mylib.c
```

Deste modo, podemos identificar qualquer erro no código de implementação ou de interface da nossa biblioteca, além de criar um arquivo de objeto biblioteca.

Criar um ARQUIVO DE OBJETO COMPARTILHADO

Como forma alternativa, você pode optar por criar um arquivo de objeto compartilhado de um ou mais arquivos `.o` que por sua vez, podem ser utilizados em outros programas que usam a sua biblioteca. Um arquivo de objeto compartilhado (*Shared Object file*) é o nome Unix para uma biblioteca ligada dinamicamente no qual o código é carregado no arquivo `a.out` - esse é nome padrão para o arquivo executável gerado após compilação do código do programa utilizando o gcc - a tempo de compilação. Para criar um arquivo `.so` deve-se utilizar a bandeira `-shared` no gcc. Um exemplo de como deve ser colocado no terminal do seu compilador é este:

```
$ gcc -o mylib.o -c mylib.c
$ gcc -o anylib.o -c anylib.c
$ gcc -o otherlib.o -c otherlib.c
$ gcc -shared -o libnome_bib.so mylib.o anylib.o otherlib.o -lm
```

Onde `mylib.o`, `anylib.o` e `otherlib.o` são exemplos de arquivos de objeto biblioteca, e `lib_nome_bib.so` é um exemplo para o nome do arquivo de objeto compartilhado que une a três bibliotecas citadas. O `lib` antes de `lib_nome_bib.so` é necessário pois o gcc assume que toda biblioteca começa com `lib` e termina com `.so` ou `.a`.

Criar uma FILA DE ARQUIVOS

Você também poderia criar uma fila de arquivos (uma biblioteca ligada estaticamente) de um ou mais arquivos `.o`. Se você ligar a uma biblioteca estática, tal código será copiado para o arquivo `a.out` a tempo de execução.

USAR a biblioteca em outros códigos em C/C++

Para utilizarmos a nossa biblioteca em outros arquivos devemos primeiramente adicionar a linha de código `#include "mylib.h"` no nosso código principal. Logo em seguida devemos escolher uma das três opções, de acordo com o tipo de biblioteca que estamos trabalhando:

cod:compgcc

```
//Ligando o programa principal com um arquivo de objeto biblioteca
$ gcc main.c mylib.o

//Ligando o programa principal com um arquivo libnome_bib.so (ou .a)
$ gcc main.c -lnome_bib

//Se a biblioteca nao estiver no caminho de arquivo padrao
$ gcc main.c -L/ieee/cs/bibliotecas -lnome_bib
```

Listing 6.3: Exemplos de como compilar um código que está utilizando uma biblioteca criada pelo programador.

Executando um arquivo executável ligado a um arquivo de objeto compartilhado

Se o arquivo de objeto compartilhado não estiver na aba `/usr/lib` do seu explorador de arquivos, você precisará configurar a sua variável de ambiente `LD_LIBRARY_PATH` para então o vinculador de tempo de execução (*runtime linker*) possa achar e carregar o seu arquivo `.so` no arquivo executável em tempo de execução:

cod:exec.so

```
//Se estiver utilizando o bash:
export LD_LIBRARY_PATH=/ieee/cs/lib:$LD_LIBRARY_PATH
//Se estiver utilizando o tcsh:
setenv LD_LIBRARY_PATH /ieee/cs/lib:$LD_LIBRARY_PATH
```

Listing 6.4: EXECUTANDO um arquivo executável ligado a um arquivo de objeto compartilhado.

Gostaríamos de aproveitar essas últimas linhas do nosso capítulo para orientá-los a darem preferência ao arquivo de objeto na hora de compilar suas bibliotecas. Primeiro porque são mais simples de entender e de compilar (a níveis humanos, não a nível computacional), segundo porque a diferença para vocês que estão começando é mínima, ou seja, não justifica a utilização de arquivos `.so` e `.a`, sendo esses possíveis agentes contrários ao seu aprendizado.

A ideia principal desse capítulo foi expor as principais bibliotecas em C/C++, bem como ensinar como criar e utilizar uma biblioteca feita por você do zero absoluto.

Sendo esse o último capítulo da nossa apostila, nós da CS UFRN gostaríamos de parabenizá-lo por ter terminado a leitura da primeira apostila de C/C++. Esperamos ansiosamente pelo seu contato para esclarecimento de qualquer tipo de dúvida sobre a apostila ou o conteúdo abordado por ela. Gostaríamos também de pedir que enviasse um feedback para o nosso email cs.ieee.ufrn@gmail.com e nos siga no instagram @csufrn. Não se esqueça de dar uma olhadinha no desafio final no Capítulo 1 da Parte III desta apostila. Até a próxima!

PARTE III

Desafio Final

CAPÍTULO 1

O famoso e icônico Jogo da Velha

E para ter a certeza de que a apostila foi proveitosa, nossa equipe acredita que nada é melhor que a colocar a mão na massa. Portanto, dedicamos essa terceira parte para esse exercício de fixação que pode até parecer simples, mas que abordará todo o assunto que estudamos até aqui.

O desafio final consiste na elaboração de um jogo da velha para 2 jogadores. Tal jogo deve indicar a vez do jogador e dizer quando e qual jogador foi campeão. Se não houver campeão, o programa deve indentificar o empate e exibir a mensagem "Deu velha!". É importante que o programa imprima o tabuleiro todas as vezes que o jogador 1 ou jogador 2 vá escolher em qual casa jogar, bem como no término da partida. Além disso, seu programa deve estar **à prova de erros**, isto é, caso o jogador 1 ou jogador 2 digite algo inesperado ou escolha uma casa que já foi escolhida, o seu programa não pode parar! Ele deve ser esperto o suficiente para reconhecer possíveis erros do usuário e retornar mensagens no prompt como "Ops! Você digitou algo inesperando!" e permanecer na vez do jogador que digitou algo errado.

A resolução do exercício pode ser consultada no nosso repositório aberto do Github, sob o nome de `minicurso_CCpp`.

Bibliografia

funcC	[Cas]	Casavella, E. <i>Funções em C</i> . URL: http://linguagemc.com.br/funcoes-em-c/ . (acessado: 09.09.2019).
streams	[cpl]	cplusplus. <i>Input/Output</i> . URL: http://www.cplusplus.com/reference/iolibrary/ . (acessado: 20.04.2020).
CreatebibCCpp	[Dep]	Department, S. C. C. S. <i>C Libraries</i> . URL: https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_C_libraries.html . (acessado: 23.04.2020).
Cbib	[Feo]	Feofiloff, P. <i>Bibliotecas de funções</i> . URL: https://www.ime.usp.br/~pf/algoritmos/apend/interfaces.html . (acessado: 23.04.2020).
repeticaoPuc	[Pina]	Pinho, M. S. <i>Comandos de Repetição</i> . URL: https://www.inf.pucrs.br/~pinho/Laprol/ComandosDeRepeticao/Repeticao.html . (acessado: 12.09.2019).
lingCppIO	[Pinb]	Pinho, M. S. <i>Entrada/Saída com Streams</i> . URL: https://www.inf.pucrs.br/~pinho/PRGSWB/Streams/streams.html . (acessado: 01.07.2019).
dadosTutPoint	[Poi]	Point, T. <i>C++ data types</i> . URL: https://www.tutorialspoint.com/cplusplus/cpp_data_types.htm . (acessado: 01.07.2019).
typeCasting	[pro]	programmer, i'm. <i>Type Casting In C Language</i> . URL: https://www.improgrammer.net/type-casting-c-language/ . (acessado: 20.04.2020).
quoraVar	[Quo]	Quora. <i>What are the variables in programming languages?</i> URL: https://www.quora.com/What-are-the-variables-in-programming-languages . (acessado: 01.07.2019).
repeticaoDev	[San]	Santos, A. <i>Estrutura de Repetição: C++</i> . URL: https://www.devmedia.com.br/estrutura-de-repeticao-c/24121 . (acessado: 12.09.2019).
launchVar	[Sch]	School, L. <i>What is a variable</i> . URL: https://launchschool.com/books/ruby/read/variables . (acessado: 01.07.2019).
lingCInOut	[Tre]	Treinamento, I. T. e. <i>Operações de entrada e saída de dados em Linguagem C</i> . URL: http://linguagemc.com.br/operacoes-de-entrada-e-saida-de-dados-em-linguagem-c/ . (acessado: 01.07.2019).
passArrFunc	[Tut]	TutorialsPoint. <i>C++ Passing Arrays to Functions</i> . URL: https://www.tutorialspoint.com/cplusplus/cpp_passing_arrays_to_functions.htm . (acessado: 13.04.2020).
w3schools	[w3s]	w3schools. <i>C++ If-Else</i> . URL: https://www.w3schools.com/cpp/cpp_conditions.asp . (acessado: 10.09.2019).
Cpbib	[Wik]	Wikipédia, a. e. l. <i>Biblioteca padrão do C++</i> . URL: https://www.wikiwand.com/pt/Biblioteca_padr%C3%A3o_do_C%2B%2B . (acessado: 23.04.2020).