

Manual da Linguagem C++: Do Zero à Computação Paralela (*Draft*)

Antônio Igor Cavalcante Lima
Elthon Alex da Silva Oliveira
Samuel Albuquerque

Universidade Federal de Alagoas - *Campus* Arapiraca
Fevereiro de 2017

DRAFT

Apresentação

Olá, este manual foi produzido com o intuito de auxiliar alunos que estão começando a entrar no enorme e fascinante universo da computação. Sendo assim, começo do nível mais básico possível, para atender pessoas que nunca viram programação, e seguirei, aos poucos, até chegar em um assunto relativamente complexo que é a Programação Paralela.

Mas calma! Sem desespero! Antes de se assustar com tantos termos estranhos, tente estudar tudo do início e com calma que vai dar tudo certo: usarei uma linguagem levíssima, descontraída e sempre com exemplos bem didáticos pra que todos possam entender direitinho. O principal diferencial desse manual é que foi escrito tentando fazer com que você sinta prazer em ler coisas que muitas vezes são escritas de forma chata.

Iniciarei explicando conceitos super básicos como os de algoritmo e funcionamento do computador, explicarei como configurar ambientes de desenvolvimento, como fazer programas dos mais simples até alguns mais complicados e, depois de construir uma base sólida, usarei um projeto desenvolvido na UFAL Arapiraca como estudo de caso para exercitar os aprendizados com algoritmos e introduzir o assunto de Programação Paralela, que trata basicamente de dividir uma operação complexa em várias menores e executar todas ao mesmo tempo em mais de um processo.

Todos os códigos exibidos neste manual, as respostas dos exercícios, e algumas outras coisas que julgar importantes estarão no meu GitHub disponíveis para download:

<https://github.com/igorcaavalcante/>.

Este manual está sendo distribuído gratuitamente, mas como recompensa gostaria de te ouvir, saber se te ajudou, se precisa de alguma correção ou algo assim. Você pode me contatar através do meu email (igorcavlim@gmail.com) ou de redes sociais (geralmente uso o nick [igorcaavalcante](#)) tipo Instagram e Facebook. Espero de verdade que você aprenda e se divirta lendo este manual, o mundo da programação é muito massa. :)

Agradecimentos

DRAFT

Sumário

I	Conceitos Iniciais e Linguagem C++	7
1	Conceitos básicos do básico	8
1.1	Coisas que você precisa saber antes de tudo	8
1.2	<i>Hardware</i> e <i>Software</i> : O que são e como interagem	8
1.3	O que é um algoritmo?	9
1.4	Uma rápida explicação sobre como os programas são executados	10
1.5	Linguagens de programação	11
1.6	Como faremos NOSSOS programas?	12
1.7	Nosso primeiro algoritmo	12
1.8	Um rápido exercício	13
2	Preparando e conhecendo o ambiente de desenvolvimento	14
2.1	Baixando e instalando o Dev C++	14
2.2	Iniciando nosso primeiro projeto	15
3	Finalmente, programas!	17
3.1	Olá Mundo!	17
3.2	Variáveis	20
3.2.1	Variáveis em C++	21
3.2.2	Tipos de dado	24
3.2.2.1	Algumas observações	24
3.2.3	Variáveis na prática	25
3.2.4	Modificadores de tipo	26
3.2.4.1	Modificadores de tipo na prática	27
3.2.5	Alguns cuidados necessários	29
3.2.6	Lendo dados	29
3.2.7	Operadores aritméticos	29
3.2.8	Chars: letras e palavras (EM CONSTRUÇÃO)	32
3.3	Constantes	34
4	Estruturas de seleção e de repetição	36
4.1	Tomando decisões: estruturas de seleção	36
4.1.1	<i>if...else...</i>	36

4.1.1.1	Operadores comparativos (relacionais)	38
4.1.1.2	Operadores Lógicos	39
4.1.1.3	Exemplo prático: par ou ímpar?	39
4.1.1.4	Operador ternário	41
4.1.2	<i>switch...case...</i>	41
4.1.3	Como usar as chaves	43
4.2	<i>Loops</i> : estruturas de repetição	44
4.2.1	<i>do...while</i>	46
4.2.2	<i>while</i>	48
4.2.3	<i>For</i>	50
4.3	Funções	52

II Caso de Uso: Simulação de Sistemas Usando Autômatos Celulares e C++ 53

III Paralelizando o Processamento dos Dados com MPI 54

Parte I

Conceitos Iniciais e Linguagem C++

Capítulo 1

Conceitos básicos do básico

1.1 Coisas que você precisa saber antes de tudo

Mesmo que contenha tópicos avançados como o de programação paralela, o público alvo deste manual é uma pessoa que nunca viu programação na vida. Assim, terei que explicar conceitos muito básicos como o de algoritmos, armazenamento de dados, IDEs, etc. Tais conceitos podem facilmente entender alguém que já possui noções básicas de programação. Mesmo que eu recomende fortemente que todos leiam este manual por completo, caso você já tenha uma boa base dos conhecimentos citados anteriormente e se sentir entediado(a) ao lê-los de novo, pode pular para a seção 2, na qual ensinarei a instalar os programas necessários para começarmos de fato a brincadeira.

1.2 *Hardware e Software*: O que são e como interagem

Atualmente, estamos rodeados por aparelhos eletrônicos como *smartphones*, *tablets*, *notebooks*, etc. Para serem úteis, esses objetos que precisam processar dados (fazer cálculos) durante sua utilização. Logo, podemos considerar que são computadores (o que computa, calcula). Sendo computadores, precisam de programas (*software*) que façam a manipulação dos dados obtidos do mundo externo por meio dos componentes físicos (*hardware*).

Para facilitar o entendimento, pegaremos nosso bom e velho telefone celular como exemplo. Para que você possa fazer uma ligação, o seu telefone precisa ser capaz de obter as coisas que você fala (através do microfone, que é *hardware*), traduzir as palavras de forma que os componentes internos compreendam, fazer algumas dezenas de operações para que só depois possa ser feita essa comunicação. Tudo isso instantaneamente.

Todo esse rodeio pra dizer que computadores trabalham com *hardware* e *software*, sendo o *hardware* o conjunto de componentes físicos (tangíveis) que obtém dados do mundo físico e *software* a parte computacional, a interface que faz a comunicação entre nós e as máquinas, o programa que transforma os dados obtidos em informação ¹.

¹Entenda informação como o resultado do processamento de dados de forma que se tornem úteis pra nós. Exemplo: dados obtidos por um sensor de temperatura se tornam informação quando são processados e nos apresentados

1.3 O que é um algoritmo?

O que nos interessa nesse manual são os programas. Anteriormente já expliquei que um programa é um *software*, mas, mais e melhor que isso, um programa é, em sua essência, um **algoritmo**. Essa palavrinha que você provavelmente ainda não conhecia será nosso objeto de estudo durante todo o manual e será o conceito mais importante e fundamental que veremos. De forma bem simples:

Algoritmo é um conjunto de instruções organizadas de forma lógica e sequencial que visam resolver um problema.

Refletindo sobre isso, entende-se que esse conceito não é restrito ao universo da computação. Todo problema, seja qual for sua natureza, necessita de um algoritmo para que seja resolvido. Estar com fome, por exemplo, é um problema recorrente em minha vida. Para isso, necessito de um algoritmo que resolva isso pra mim. Um que resolve muito bem meu problema é o algoritmo do bolo (reza a lenda que se o exemplo da receita de bolo não for dado no ensino de algoritmo, a pessoa terá sete anos de azar):

Ingredientes: 2 xícaras(chá) de açúcar, 3 xícaras (chá) de farinha de trigo, 4 colheres (sopa) de margarina, 3 ovos, 1 e 1/2 xícara (chá) de leite, 1 colher (sopa) bem cheia de fermento em pó.

Modo de preparo:

- 1 - Bata as claras em neve e reserve
- 2 - Misture as gemas, a margarina e o açúcar até obter uma massa homogênea
- 3 - Acrescente o leite e a farinha de trigo aos poucos sem parar de bater
- 4 - Por último, adicione as claras em neve e o fermento
- 5 - Despeje a massa em uma forma grande de furo central untada e enfarinhada
- 6 - Asse em forno médio à 180 °C (preaquecido) por aproximadamente 40 minutos ou até furar com um garfo e ele sair limpo

Pronto, esse é um algoritmo que resolve -e muito bem resolvido- o problema da nossa fome. Observe que o modo de preparo tem todas as características de um algoritmo que já foram citadas: é uma lista de instruções que resolve um problema e possui uma ordem lógica, afinal, se a criatura alterar a ordem especificada ou deixar de fazer alguma o resultado provavelmente não vai ser muito bom.

Outro algoritmo bem comum e mais próximo da computação é o da calculadora. Instintivamente provavelmente pensaremos que somar dois números pode ser descrito como: “escolha dois números e mostre a soma”. Só que na programação as coisas precisam ser um pouco mais detalhadas. Dessa forma, uma calculadora poderia ser descrita assim:

em graus Celsius.

- 1 - Escolha o primeiro número
- 2 - Escolha a operação
- 3 - Escolha o segundo número
- 4 - Realize a operação
- 5 - Mostre o resultado

Ou seja, bem mais detalhado do que pensaríamos pela primeira vez. Mais pra frente iremos utilizar este mesmo algoritmo pra fazer nossa calculadora e, acreditem, ainda temos várias outras instruções a acrescentar, mas por hora esse já está bom.

1.4 Uma rápida explicação sobre como os programas são executados

Futuramente, ao entrar na programação, considero que seja importante entender como um computador funciona. Então vou tentar fazer isso de uma forma simples e rápida sem me prender a alguns detalhes, então, caso tenha interesse em se aprofundar mais no assunto, procure um livro sobre Arquitetura e Organização de Computadores que vai te explicar em detalhes estes processos que quero introduzir.

Os programas instalados no seu computador ficam guardados no seu *Hard Drive*, que em português significa "disco rígido" e é popularmente conhecido apenas pelas iniciais **HD**. Explicando de forma bem superficial, HD é caracterizado pela capacidade de guardar uma grande quantidade de dados e pela capacidade de mantê-los mesmo depois de o computador ser desligado, ou seja, é uma memória do tipo não volátil. O único problema do HD é que, apesar de parecer muito pra humanos, ler dados a 7200rpm é muito lento para o computador. Por esse motivo, o HD é considerado uma memória secundária, sendo a primária a Memória RAM.

A memória RAM, *Random Access Memory* (em português Memória de Acesso Aleatório), por sua vez, é uma memória cujo tempo de leitura é muito superior ao do HD, mas, em contrapartida, ela não consegue armazenar muitos dados e os perde quando fica sem energia.

Por último e mais importante, o processador. Ele é o centro do computador: é responsável por todo o processamento dos dados, que consiste basicamente em fazer contas matemáticas, uma vez que o computador só entende os números 0 e 1². Ele pega o programa (as vezes parte dele) no HD e coloca na memória RAM. Só agora posso explicar a principal função da RAM: armazenar os dados dos programas em execução, já que ela tem uma velocidade de acesso muito maior³ que o HD. Como o computador tem milhares de processos abertos (todos na memória RAM⁴), ele pega um, processa dados, devolve pra RAM, pega outro, processa, devolve, etc. Tudo isso milhares de vezes, por isso é tão importante ter uma memória com rápido tempo de acesso.

²Para saber mais sobre isso, dá um Google sobre Sistema Binário, pois no momento não é meu foco detalhar isso.

³Apenas a título de curiosidade, existem memórias mais rápidas que a RAM. Pesquise por Pirâmides de Memórias que existem imagens bem didáticas sobre o assunto.

⁴O HD também pode armazenar programas em execução, mas como novamente não é nosso foco, caso queira saber mais, dá um Google sobre Memória Virtual.

Dito tudo isso, para exemplificar e amarrar todo o conteúdo, imagine que você quer abrir a calculadora. Assumindo que ela já está instalada no seu computador, o processador recebe seu pedido e manda buscar-la no seu HD, afinal é lá que ela fica instalada. Após carregar na RAM, abre aquela janelinha e fica aguardando o usuário digitar um número. Após digitar o número, o guarda na RAM, espera a operação, faz o mesmo com ela e fica esperando o segundo número. Adivinha o que ele faz? Guarda na RAM também! Depois, quando você clica no sinal de igualdade, ele pega os dois números, realiza a operação e -pasmem!- guarda na RAM! Só depois ele busca o resultado e nos mostra. Tudo isso numa fração de segundo! É um assunto bem complexo, mas a parte que nos interessa é essa mais superficial mesmo, não se espantem! Caso não tenha ficado muito claro essa última parte, aguarde que quando fizermos uma calculadora tudo vai fazer sentido e vai ficar bem trivial.

1.5 Linguagens de programação

Como já expliquei, programas são algoritmos, e algoritmos são basicamente um conjunto de comandos para que o computador execute algumas ações que resolvam um problema. Dito isso, caso queiramos que o computador some dois números é só escrever “algumas coisas” e mandar ele executar. Mas não é assim tão simples, não basta escrever “Computador, some estes dois números!” pois é óbvio que o computador não vai entender. É pra isso que existem as linguagens de programação, uma vez que são:

normas para organizar, sintática e semanticamente, instruções que o computador seja capaz de entender e realizar ações a partir desses comandos.

Já que o computador não consegue nos compreender do jeito que queríamos, as linguagens fazem esse meio termo pra nós. Mas pense comigo: um computador é um dispositivo eletrônico que só entende o sistema binário, certo? Certo. Então precisamos aprender o sistema binário e fazer tudo em binário?? Graças a Deus, NÃO :). Alguns seres iluminados antigamente definiram comandos que nós conseguimos entender e desenvolveram programas que transformam o que a gente entende em uma linguagem que a máquina entende (as chamadas **linguagem de máquina**)! Dito isso, costuma-se classificar linguagens em uma espécie de ranking: linguagens de alto ou baixo nível. Isso quer dizer o seguinte: quanto mais baixo for o nível da linguagem, mais perto ela é daquela que a máquina entende. Obviamente, quanto mais alto o nível, mais próximo do que nós entendemos (para nossa alegria, C++ é de alto nível).

Assim, ao nos depararmos com um problema, pensamos em uma forma de resolvê-lo (começamos a imaginar um algoritmo) e logo depois pensamos em uma linguagem que resolva da melhor forma esse problema (uma linguagem não é capaz de fazer tudo, cada uma tem foco em um tipo de problema). Faremos uso da linguagem C++, mas tenha em mente que várias outras linguagens poderiam ser escolhidas para fazer o que faremos.

1.6 Como faremos NOSSOS programas?

Agora a coisa tá ficando boa! Depois de tantos conceitos introdutórios, vamos falar do que realmente interessa: NOSSOS PROGRAMAS.

Programaremos em C++, uma linguagem de alto nível que já é bem tradicional na nossa área e que nasceu a partir da linguagem C, tendo como principal característica a introdução da programação orientada a objetos (assunto que não trataremos mas que vale a pena ser pesquisado depois de ter lido este manual).

Para isso, nossas instruções serão escritas em um documento de texto salvo com a extensão .cpp, que identifica pro computador que aquilo é o **código-fonte ou arquivo-fonte** (forma como chamamos um arquivo de comandos que posteriormente virará um programa) de um programa em C++. Logo, caso crie uma calculadora, poderá chamar seu código fonte de *calculadora.cpp* ou qualquer outro nome, desde que tenha a extensão .cpp, claro. Sendo assim, poderíamos usar alguns processadores de texto como o bloco de notas no *Windows* (*Word* é editor, não serve pra isso!) ou o Gedit nos ambientes Linux para escrever nossos programas. Porém, lembra que eu te falei que uns seres abençoados desenvolveram programas que convertem os nossos em linguagem de máquina? Pois bem, esses programas são chamados de **compiladores**⁵. Sendo assim, após nós escrevermos nossos programas, devemos mandar os compiladores transformarem nossos algoritmos em programas de fato. Só que usar o bloco de notas e depois mandar compilar é meio chato de fazer e já já você vai entender que o bloco de notas não nos ajuda a programar. Pensando nisso, foram criadas as **IDEs** (do inglês *Integrated Development Environment* ou Ambiente de Desenvolvimento Integrado), que são programas que geram pra gente uma interface de desenvolvimento que facilitam -e muito- nossa vida. Elas possuem um editor de texto que diferencia por cores nossos comandos e nos mostra exatamente nossos erros (acredite que isso é uma coisa ótima, pois encontrar a falta de uma vírgula em um programa de 1000 linhas sem ajuda de uma IDE não é uma tarefa fácil!). Além disso, uma IDE nos permite compilar e executar nosso programa com apenas um clique, coisa que tomaria mais alguns passos caso fôssemos fazer sem ela. Neste manual, usaremos o **Dev-C++ no Windows**, mas também podem usar o *Code::Blocks*, que funciona tanto no *Windows* quanto no Linux e tem funcionalidades bem parecidas.

1.7 Nosso primeiro algoritmo

Antes de partir pra programação de verdade, vamos fazer um algoritmo super simples feito de forma beeeem informal pra que depois seja mais fácil de você assimilar os conceitos. Será como se nós estivéssemos realmente conversando com um computador e explicando a ele como ele vai fazer isso pra gente.

Antes de ler a resposta abaixo, passe uns minutos pensando (de preferência escreva!) em um algoritmo que calcule a média das 4 notas de um aluno e verifique se ele foi aprovado (média maior ou igual a 6) ou reprovado

⁵Infelizmente, não entrarei em detalhes sobre compiladores e o processo de compilação por ser um assunto extenso, mas recomendo fortemente que você pesquise sobre isso!

Esperando do fundo do meu coração que você tenha tentado fazer, um algoritmo pra isso ficaria mais ou menos assim:

1. Peça as 4 notas;
2. Guarde as 4 notas;
3. Some as 4 notas e depois divida por 4. Guarde o resultado desta operação e chame-o de “média”;
4. Caso a média tenha sido maior ou igual a 6, informe que o aluno foi aprovado;
5. Caso a média tenha sido menor que 6, informe que o aluno foi reprovado.

Confie em mim: esse pseudocódigo (algoritmo escrito em linguagem natural de forma que qualquer pessoa entenda) é bem próximo do que faremos quando estivermos programando. Inclusive, mais tarde faremos um programa assim e usaremos esse pseudocódigo como base.

1.8 Um rápido exercício

Agora é com você: pra exercitar o que aprendeu, escreva um pseudocódigo que leia dois números e informe qual é o maior dos dois.

Capítulo 2

Preparando e conhecendo o ambiente de desenvolvimento

2.1 Baixando e instalando o Dev C++

Como já disse anteriormente, usarei uma IDE chamada Dev C++ para nossos programinhas no sistema operacional Windows (usei o 8, mas acredito que funciona até no XP). Baixei o meu Dev C++ nesse *link*: <https://sourceforge.net/projects/orwelldevcpp/> mas, caso esse *link* não funcione mais, jogue no pai Google e se desenrole, não é difícil de achar não. Atentem pra versão: estou usando a 5.11.

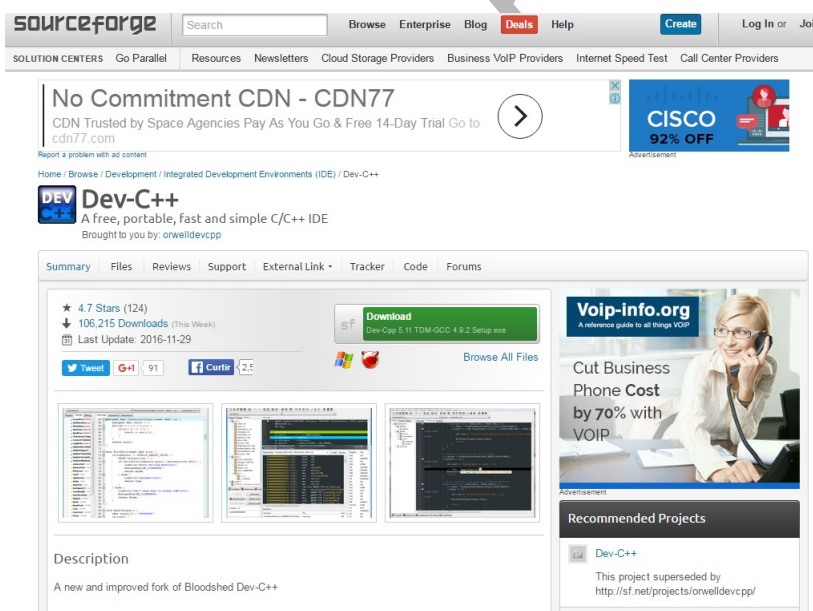


Figura 2.1: Clique no botão verde pra baixar, simples assim. Fonte: Autor

Depois de baixado, pra instalar também é bem simples:

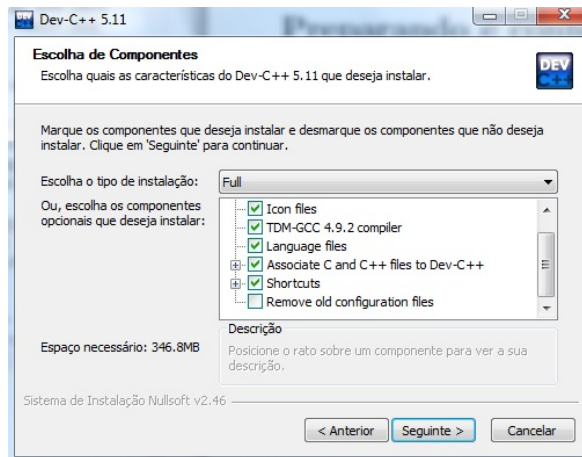


Figura 2.2: Detalhe apenas para o tipo de instalação, pode escolher a *Full* (completa) mesmo. Fonte: Autor

Assim que você instalar, verá uma tela pra escolha do idioma. Recomendo que escolham Alemão só pelo desafio (brincadeira, pode escolher Português mesmo). Logo depois ele te oferece algumas opções visuais e a escolha fica por sua conta (inclusive tem um visual estilo aqueles códigos de filme que é massa pra pagar de hacker).

2.2 Iniciando nosso primeiro projeto

Finalmente, tudo instalado e configurado, vamos iniciar um novo (e primeiro) projeto. Lembra que eu disse anteriormente que uma IDE facilita muito nossa vida? Então, quando criamos um projeto ela prepara tudo pra gente e já deixa prontinho pra a gente começar a desenvolver nosso programa. Ou seja, ela cria um novo arquivo do tipo .cpp (que é próprio para programas em C++) e já deixa aberto pra gente começar. Ela cria outros arquivos também, mas vamos deixar pra depois, por enquanto.

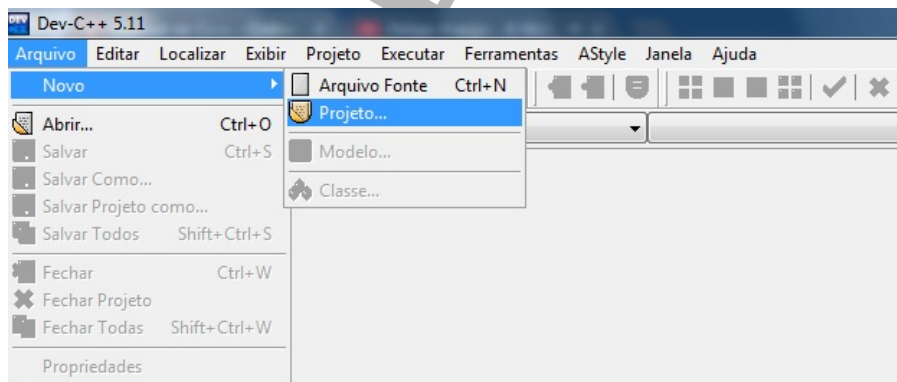


Figura 2.3: Esta é a interface do programa sem um projeto aberto. Inicie um novo da forma mostrada acima. Fonte: Autor

Ao iniciarmos um novo projeto, ele nos oferece as seguintes configurações:

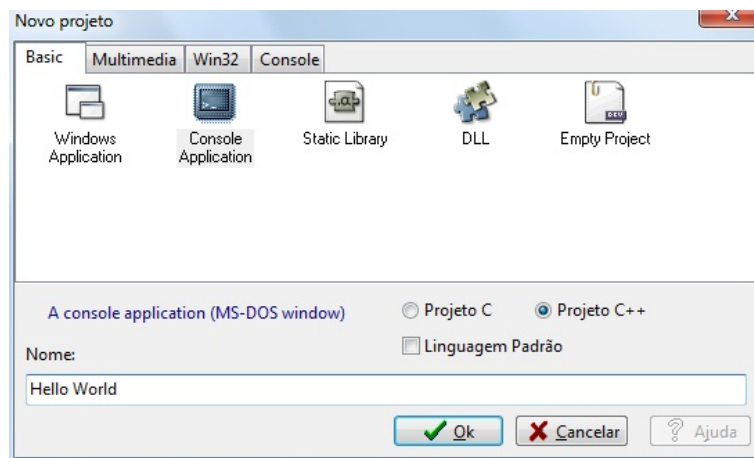


Figura 2.4: Configurações de novos projetos. Siga esse modelo pra configurar o seu. Fonte: Autor

Traduzindo: “*Console Application*” significa que nosso programa será rodado em uma janelinha do prompt de comando, “Linguagem C++” obviamente é a linguagem utilizada e “Nome” é o nome do projeto.

Depois disso, basta escolher um local pra salvar o projeto e veremos nosso primeiríssimo programa, que na verdade é só um “esqueleto” de programa: ainda teremos que adicionar alguns comandos para que ele de fato faça algo.

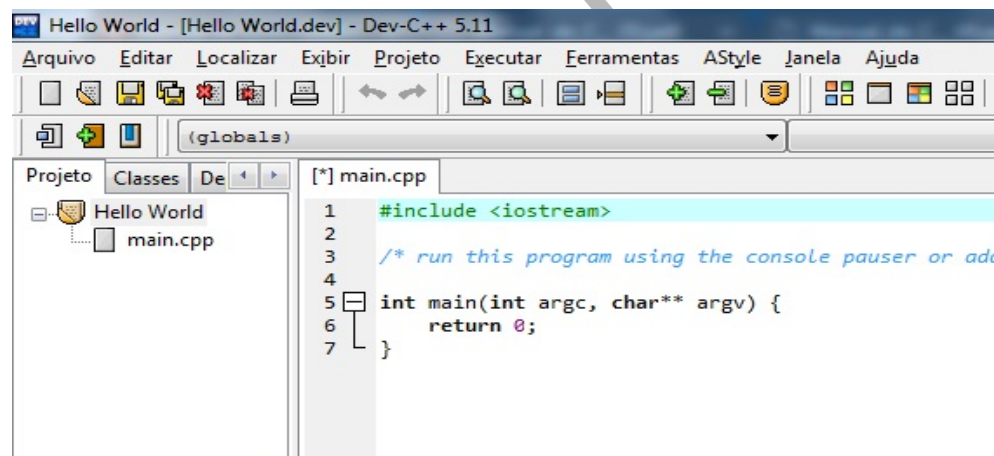


Figura 2.5: “Esqueleto” de um programa em C++. Fonte: Autor

Aproveite o momento pra sair “fuçando” a IDE. Vá abrindo as abas e tentando entender algumas opções. Seja curioso e vai aprender muito! Agora vamos pra parte mais legal: a prática!

Capítulo 3

Finalmente, programas!

3.1 Olá Mundo!

Agora entraremos pra valer na programação! É tradição no mundo da computação começar o aprendizado de todas as linguagens com um **Hello World!** (“Olá Mundo!” em português). O Hello World é o programa mais simples que se pode fazer em uma linguagem e é por isso, obviamente, que sempre começamos por ele. **É importantíssimo que você execute tudo que estamos fazendo aqui, pois você só vai aprender na prática!!** Seu propósito é muito simples: mostrar a frase “Hello World!” na tela. Reza a lenda que se não fizermos um Hello World nossos programas ficarão travando por sete anos, então...



```
[*] main.cpp
1  #include <iostream>
2
3  /*
4  Este é nosso Hello World em C++.
5  Igor Cavalcante é bonito.
6  */
7
8  int main(int argc, char** argv) {
9      //Comando para mostrar a frase:
10     std::cout << "Hello World!";
11     return 0;
12 }
```

Figura 3.1: *Hello World!*. Fonte: Autor

Calma! Não se assuste ainda, vamos aos poucos que vai dar certo! Como já foi dito anteriormente, precisamos compilar este programa, ou seja, traduzi-lo para uma linguagem que o computador consiga entender e assim poder realizar as operações necessárias. Chamamos esse de **executável**, uma vez que é, de fato, seu programa. Ou seja, quando você for executar seu

programa é ele quem vai ser executado (baixo nível), e não o arquivo .cpp, que nada mais é do que o arquivo que contém os comandos escritos em C++ (alto nível). Além disso, possui extensão .exe, que é a extensão padrão do Windows para arquivos executáveis (programas). Para isso, nós clicamos em:

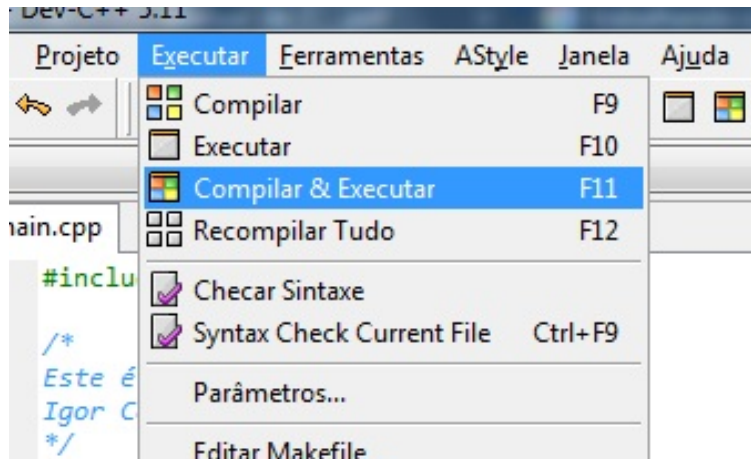


Figura 3.2: *Compilando nosso programa.* Fonte: Autor

Como você pode ver, temos algumas opções. Caso você clique em “Compilar”, a IDE vai gerar o arquivo em código de máquina e não vai fazer nada mais. Caso você clique em “Executar”, a IDE vai abrir esse arquivo que está em código de máquina (obviamente se você não tiver compilado ainda ela vai acusar erro). Caso clique em “Compilar & Executar” (que é o que eu recomendo) ela compila e logo em seguida já abre o programa pra gente, o que é bem mais cômodo e simples pra gente. Deixaremos essas outras opções pra outros momentos, mas caso queira vá mexendo aí.

Clique em “Compilar & Executar” e escolha o nome do arquivo .cpp. Ao executar, alguns antivírus como o Avast podem encarar seu programa como um vírus e tentar pará-lo, mas não se preocupem que seus programas não irão causar danos (eu espero!). Caso dê tudo certo, o resultado será este:

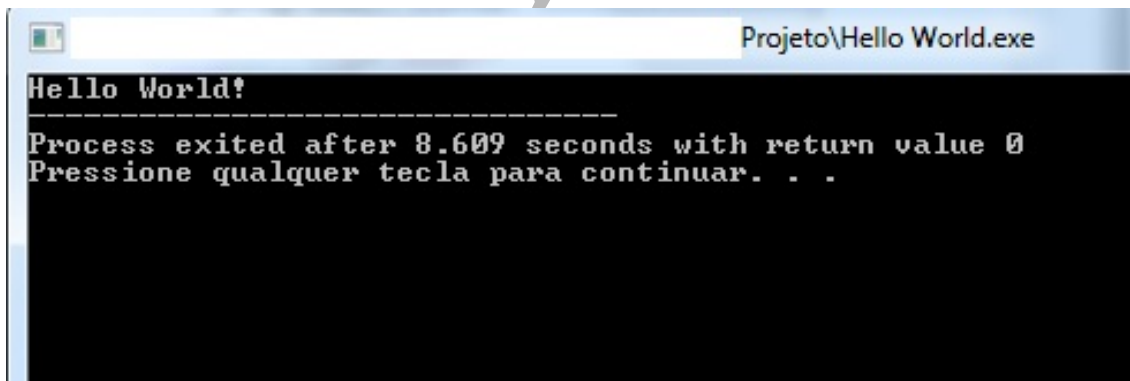


Figura 3.3: *Hello World* em execução. Fonte: Autor

Uma linguagem tem milhares funcionalidades, então seria muito custoso carregarmos todas elas em todos os programas e só usar algumas delas. Sendo assim, existem as bibliotecas, que são arquivos que contém instruções de como realizar determinadas funções. Caso a gente precise trabalhar com imagens, por exemplo, precisamos usar funções muito específicas que necessitarão de instruções de um nível mais baixo para que isso seja possível. Como nós queremos ficar no alto nível, pessoas desenvolvem essas de mais baixo e disponibilizam pra que a gente não precise reinventar a roda. Essas instruções estão disponíveis em bibliotecas. Por isso, na primeira linha do programa nós usamos o *include* (incluir) a biblioteca *iostream*¹ para que seja possível manipular a entrada e saída de dados (nesse caso só usamos a saída, que foi o texto exibido), que nos permite realizar operações de entrada (ler números, por exemplo) e saída (mostrar números, por exemplo).

A linha 8 contém a função principal do programa e sua chave de abertura. Por enquanto não se preocupe com a palavra “int”, com os parêntesis e nem com as instruções que estão dentro dele. Vamos focar em “main” e na chave de abertura ({}). Os programas em C++ são divididos em funções, que, por enquanto, tenha em mente apenas que são “blocos de código”. *Main* em português significa “principal”, logo, quer dizer que essa é a **principal função** do programa. Assim, quando o compilador² pega nosso código, ele procura imediatamente essa função pra que seja executada antes de todas outras. Observe que há um sinal de chave nessa linha e outro contrário no fim do algoritmo. Isso é o que delimita o início e fim de um “bloco de código” (poderia ter chamado de função, mas as chaves servem pra delimitar outros tipos de “blocos” que veremos mais à frente).

Informação importante: Observe que dentro do **escopo**³ da função principal eu dei um espaço (**tabulação**) da margem esquerda. O nome disso é **IDENTAÇÃO** e é **IMPORTANTÍSSIMO** que você faça isso nos seus programas toda vez que você criar um novo “bloco de código”. Caso crie um bloco dentro de outro, adicione mais um espaço (use a tecla “tab” do seu teclado). Por que eu estou dando tanta ênfase a isso? Porque **alguém vai ter que ler -e entender- seu programa!** Você trabalhará em equipe na maioria das vezes e as outras pessoas precisam entender o que você está fazendo, seu código precisa ser legível, bem organizado. Mais pra frente, quando fizermos programas mais elaborados, verá que esse espaço faz TODA a diferença no seu código!

Observe agora as linhas 3, 4, 5, 6 e 9. Além de não conterem comandos, elas obtiveram uma coloração diferente. Isso se deve ao fato de serem marcadas pelos símbolos “//” e “/**/”, que identificam **comentários** no seu programa e serão **completamente ignorados** pelo compilador, não importando seu conteúdo. Comentários servem para que você deixe explicações sobre o seu código para que depois outras pessoas -ou as vezes você mesmo- entendam o que aquele código faz. É um recurso muito útil e que sempre deve ser usado na programação pra que seu código seja legível. Quando seu comentário possuir apenas uma linha, você pode usar apenas “//” no início do seu comentário, que pode ficar no início da linha ou depois de um comando, e tudo nesta

¹*iostream*: “I” de *input* (entrada), O de *output* (saída) e *stream* significa algo parecido com fluxo. Ou seja, *iostream* = fluxo de entrada e saída

²Tenha em mente que o processo de compilação envolve outros processos, mas vou falar apenas compilador pra simplificar.

³Conceito importante: constantemente usamos esse termo pra nos referir ao contexto (“bloco de código”) em que se encontra determinado comando. Ou seja, se nos referirmos ao comando `cout`, neste caso, podemos dizer que seu escopo é o da função `main`. Caso estivesse em um “bloco” dentro do `main`, seria esse seu escopo.

linha depois disso será ignorado. Caso seu comentário possua mais de uma linha, que foi o caso do meu primeiro comentário, você pode usar “/*” e todas as linhas posteriores serão tratadas como comentários até que o compilador encontre seu fechamento, que é o “*/”. Evite comentários inúteis como o da linha 5 em seus programas, fiz apenas com fins ilustrativos (apesar de ser verdade).

Agora, vamos para o comando **cout**. Antes, observe essa notação anterior ao comando: **std::**. Ela significa que o comando cout encontra-se no **namespace** std, que é o padrão. *Namespaces* em C++ servem para resolver algumas ambiguidades que podem acontecer caso alguma biblioteca possua funções de mesmo nome. Uma analogia: digamos que existam duas Joanas na UFAL e seja preciso diferenciá-las. Para isso, ao invés de falarmos simplesmente Joana, chamamos de Ana Joana e Maria Joana. Mesma coisa acontece em C++, ao invés de escrever apenas cout e originar um possível conflito, escrevemos std::cout para informar que esse cout é do *namespace* std (que vem de *standard* e significa padrão). Posteriormente usaremos um comando para que não seja preciso digitar std:: toda vez que formos usar comandos do std, mas antes isso precisava ficar claro. Dito isso, o comando cout, de forma simples, é responsável pela saída de dados. Basta colocar o que precisa ser escrito depois dos dois sinais de menor que e tá tudo certo. Mais à frente veremos mais detalhes do cout.

Por último, temos o **return 0**, que nada mais é do que uma forma de informar ao sistema operacional que o programa funcionou corretamente e foi encerrado. Mais pra frente entenderemos melhor a função do return e entenderemos como ele está diretamente ligado ao int que fica antes de main.

A última coisa a ser dita sobre esse programa são os sinais de **ponto e vírgula (;)** que ficam no fim de **quase todas** instruções e servem pra informar onde termina a instrução. Poderíamos, então, escrever todos os comandos de um programa em uma única linha (obviamente seria horrível de entender e mais ainda de encontrar um erro) e os pontos e vírgulas fariam a diferenciação de cada comando. Não precisa colocar no fim do programa nem depois de chaves.

3.2 Variáveis

Lembra que anteriormente falei sobre como os programas são executados? Pois bem, agora alguns desses conceitos serão postos em prática. Como foi dito, os programas precisam ser alocados na nossa memória RAM e os dados que são lidos por ele também vão pra lá. Nós armazenamos os dados lidos em **variáveis**, que nada mais são do que espaços reservados na memória para salvar dados. Isso significa que, quando você digita um valor a ser somado, por exemplo, este valor fica armazenado na memória RAM pra que depois seja usado na soma.

A reserva desse espaço é chamada de **declaração de variável**. Para isso, devemos informar qual tipo de dado esse espaço vai ser reservado e também atribuir um nome a essa variável para que depois possamos acessar esse valor. Funciona igualzinho a matemática: se dissermos que X é um inteiro de valor 10 e depois perguntarmos o valor de X*2 é fácil entender que o resultado é 20. Nesse caso, o tipo da variável é inteiro, o valor é 10 e seu nome é X, simples assim. Fiz essa ilustração bem simples para que você entenda melhor:

Antes de tudo, a RAM não possui endereços em forma de números inteiros, coloquei assim apenas pra facilitar o entendimento. Além disso, coloquei reticências antes e depois para que fique

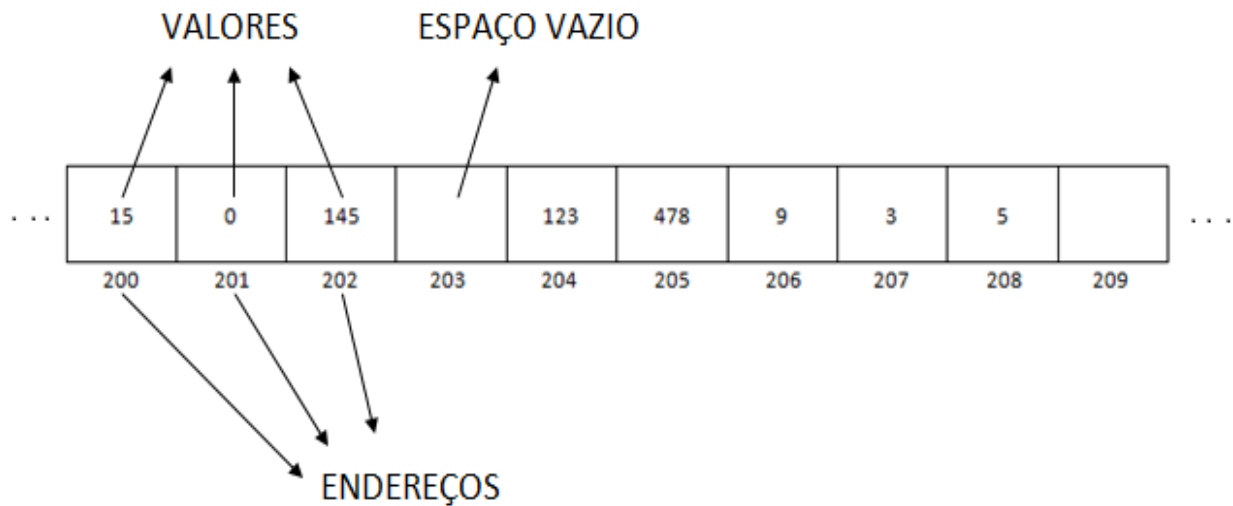


Figura 3.4: Ilustração da memória RAM. Fonte: Autor

claro que esse é um recorte: há mais espaços para os dois lados, claro.

Explicando essa imagem, é assim que podemos abstrair o funcionamento da RAM: um conjunto de espaços (representados pelos quadrados) que possui uma identificação (endereços) e que permite armazenar valores nesses espaços.

Obviamente não precisamos nos preocupar em procurar um espaço vazio ou decorar os endereços de memória: o sistema operacional faz isso pra gente. Seguindo o exemplo anterior do dobro de 10, quando declarássemos a variável `X` ele procuraria um espaço de tamanho suficiente (sobre esses tamanhos, veja a seção de Tipos de Dado) e definiria que o endereço 203, por exemplo, conteria o valor de `X`.

Pense na variável como uma caixa de um determinado tamanho que é capaz de guardar valores compatíveis com esse tamanho. Para acessarmos estes valores, basta chamar a caixa pelo nome que você colocou nela.

3.2.1 Variáveis em C++

Partindo para a prática e continuando no exemplo do dobro, poderíamos fazê-lo de inúmeras formas, vou mostrar algumas delas. Veja na Figura 3.5 uma forma didática e detalhada de como fazer isso.

Logo de cara, observe que não estou mais usando o `std::` no comando `cout`. Isso se deve ao fato de que na linha 8 eu defini que o `namespace` a ser usado na função `main` era o `std`. Ou seja, tudo que for usado nessa função se refere ao `std`, então podemos omiti-lo. Caso quisesse usar para o programa inteiro, bastaria colocar esse comando antes da função `main`. **Mas fique sabendo:** usar esse `namespace` em todo o programa não é lá uma boa prática, pois caso use mais algumas bibliotecas, pode gerar um erro no seu programa por questões de ambiguidade que foram citadas anteriormente.

Outra coisa: veja que deixei algumas linhas em branco ao longo do código. Fiz isso pra deixar

```

[*] main.cpp
1  #include <iostream>
2
3  /*Este programa serve pra exemplificar o
4  uso de variáveis mostrando o dobro de 10.*/
5
6  int main() {
7      //Definindo o namespace:
8      using namespace std;
9
10     //Declarando a variável inteira com valor 10:
11     int x = 10;
12
13     //Declarando uma variável que conterà o resultado do dobro:
14     int dobro;
15
16     //Atribuindo um valor para essa variável:
17     dobro = x*2;
18
19     //Mostrando seu dobro:
20     cout << "O dobro de 10 é " << dobro;
21
22     return 0;
23 }

```

Figura 3.5: Primeira e mais detalhada forma de se mostrar o dobro de 10. Fonte: Autor

os comandos mais separados e, assim, deixá-los mais simples de enxergar. O compilador irá ignorar linhas em branco, então você pode usá-las quando achar que serão necessárias para melhorar o entendimento do algoritmo.

Continuando, na linha 11 nós declaramos a variável `x` do tipo `int` (`int` é de **inteiro**, veremos na próxima seção os tipos de dados) e atribuímos a ela o valor 10. Na 14 declaramos outra variável que posteriormente guardará o valor do dobro, na 17 guardamos o dobro na variável declarada e na 20 mostramos o resultado.

Poderíamos ter colocado a multiplicação logo na declaração da variável `dobro` e omitido o comando da linha 17, mas estou mostrando formas de se fazer a mesma coisa. Cada caso é um caso, podem haver situações em que é melhor colocar a operação na declaração e, em outros, colocar separada.

Além disso, poderíamos ter declarado as duas variáveis em uma única declaração (desde que, é claro, sejam do mesmo tipo). Basta apenas separá-las por vírgula:

```
int x = 10, dobro;
```

Observe que eu declarei a variável `x` e ao mesmo tempo atribui um valor a ela, mas na declaração da `dobro` eu não coloquei valor. Isso está correto, mas tenha uma coisa em mente: **se você deixar a variável sem valor e tentar acessá-la o sistema operacional vai mostrar um valor aleatório chamado lixo de memória**. É como se o sistema não permitisse deixar variáveis vazias, então coloca um valor qualquer.

Confira na Figura 3.6 outra uma forma mais “enxuta” de escrever o mesmo programa. Dessa vez quis mostrar que você pode colocar várias frases ou valores no mesmo `cout`, basta colocar os

```

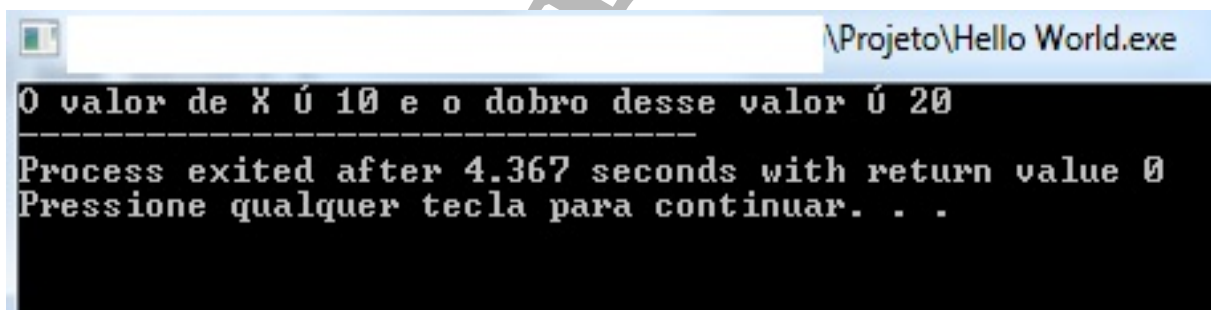
main.cpp
1  #include <iostream>
2
3  /*Este programa serve pra exemplificar o
4  uso de variáveis mostrando o dobro de 10.*/
5
6  int main() {
7      //Definindo o namespace:
8      using namespace std;
9
10     //Declarando a variável inteira com valor 10:
11     int x = 10;
12
13     //Mostrando seu dobro:
14     cout << "O valor de X é " << x << " e o dobro desse valor é " << x*2;
15
16     return 0;
17 }

```

Figura 3.6: Forma mais simples de mostrar o dobro de 10. Fonte: Autor

sinais de menor separando-os (lembrando que textos devem ficar entre aspas!). Além disso, o cout também mostra o resultado de operações como foi mostrado nesse exemplo com $x*2$. Sendo assim, nesse caso, a gente nem precisaria declarar uma variável pra guardar o valor 10, bastava colocar diretamente a operação no cout. Tente fazer isso sozinho e explore outras formas de fazer esse exercício.

Executando, obtemos o seguinte resultado:



```

Projeto\Hello World.exe
O valor de X é 10 e o dobro desse valor é 20
-----
Process exited after 4.367 seconds with return value 0
Pressione qualquer tecla para continuar. . .

```

Figura 3.7: Programa do dobro em execução. Fonte: Autor

Como podem ver, tivemos um problema nas letras com acentos. Isso se deve ao fato de o compilador estar configurado para o idioma inglês, que não possui acentos. Para contornar esse problema podemos usar o comando **setlocalte**, que está presente na biblioteca da linguagem C **locale.h** e informa que nosso programa encontra-se em determinado idioma (Português, nesse caso). Confira na Figura 3.8 como ficaria.

```

[*] main.cpp
1  #include <iostream>
2  #include <locale.h> //biblioteca que permite alterar o idioma
3
4  int main() {
5      using namespace std;
6
7      setlocale(LC_ALL, "Portuguese"); //definindo o idioma português
8
9      int x = 10;
10     cout << "O valor de x é " << x << " e seu dobro é " << x*2;
11
12     return 0;
13 }

```

Figura 3.8: Programa do dobro em execução usando locale. Fonte: Autor

3.2.2 Tipos de dado

Como foi dito anteriormente, cada variável possui um tipo e isso se deve ao fato de que alguns dados são “maiores” que outros. Ou seja, não há necessidade de reservar um espaço na memória de 4 bytes se você só precisará de 1 para armazenar determinado valor.

Num mundo onde as memórias RAM geralmente tem, no mínimo, uma média de 2 gigas de espaço, parece estranho falar de uma economia de míseros 3 bytes. O problema é que os programas possuem milhares de variáveis e esses míseros 3 bytes multiplicados por milhares fazem uma diferença enorme!

Conheça os tipos de dado na tabela 3.1.

Tipo	Descrição	Tamanho	Valores
char	Caracteres (“letras”)	1 byte	-128 até 127
int	Números inteiros, negativos ou positivos	4 bytes	-2.147.483.648 até 2.147.483.647
float	Números reais, negativos ou positivos	4 bytes	3.4E +/- 38 (7 dígitos)
double	Números reais, negativos ou positivos	8 bytes	1.7E +/- 308 (15 dígitos)
bool	Valores booleanos	1 byte	true ou false (1 ou 0)

Tabela 3.1: Tipos de dado. Fonte: https://www.tutorialspoint.com/cplusplus/cpp_data_types.htm. Acessado em: 12/03/2017.

3.2.2.1 Algumas observações

Char vem de **characters**, caracteres em português. Armazena UM caractere, que pode ser uma letra, um número ou alguns outros símbolos.

Observando a tabela 3.1, vemos que o tipo char pode ter valores até 127. De forma superficial, o computador trata tudo como números, mas letras não são números, certo? Então foi necessário fazer uma representação numérica para as letras e outros símbolos úteis no nosso dia a dia, e assim

nasceu a **tabela ASCII**. Ela faz a correspondência entre caracteres e números, que vão de 0 a 127. Dá um Google nisso pra conhecer que é interessante e importante. Já já veremos isso na prática.

Sobre os tipos numéricos, não há muito o que se falar. O tipo **int** guarda números inteiros, “redondos”. Já para números com casas decimais, podemos usar **float** ou **double**, cuja diferença está na **precisão**: o double, por conseguir armazenar mais casas decimais, é muito mais preciso que o float e, conseqüentemente, ocupa mais memória.

O tipo **bool** tem uma particularidade: no fundo, o tipo bool na verdade é um int com um tamanho menor, já que só precisa guardar 1 (true) ou 0 (false). Ai você pode pensar: “Já que só guardará 1 ou 0, por que não usar apenas 1 bit pra isso?”. A resposta é simples: o menor tamanho que a RAM pode endereçar é um byte, então não consegue armazenar apenas um bit.

3.2.3 Variáveis na prática

A Figura 3.9 mostra um algoritmo simples para visualizarmos as variáveis e seus tamanhos na prática.

```
[*] variaveis.cpp
1  #include <iostream>
2  #include <locale.h>
3
4  int main() {
5      using namespace std;
6      setlocale(LC_ALL, "Portuguese"); //definindo o idioma português
7
8      int inteiro = 10;
9      cout << "O número inteiro " << inteiro << " ocupa " << sizeof(inteiro) << " bytes na memória " << endl;
10
11     char caractere = 'a';
12     cout << "O caractere '" << caractere << "' ocupa " << sizeof(caractere) << " bytes na memória." << endl;
13
14     char outrocaractere = 120;
15     cout << "A variável outrocaractere guarda o caractere correspondente ao valor 120 da tabela ASCII: " << outrocaractere << endl;
16
17     float real = 0.99999;
18     cout << "O real " << real << " ocupa " << sizeof(real) << " bytes na memória." << endl;
19
20     double realmaispreciso = 0.515645154654258;
21     cout << "Já o double " << realmaispreciso << " ocupa " << sizeof(realmaispreciso) << " bytes" << endl;
22
23     bool booleano1 = true;
24     bool booleano2 = 0;
25     cout << "O booleano1 tem valor " << booleano1 << " pq é true. \nO booleano2 tem valor " << booleano2 << " pq é false. \n";
26
27     bool booleano3 = 1154;
28     cout << "Qualquer valor diferente de zero é true, o que aconteceu no booleano3: " << booleano3 << endl;
29     return 0;
30 }
```

Figura 3.9: Programa que exemplifica o uso das variáveis Fonte: Autor

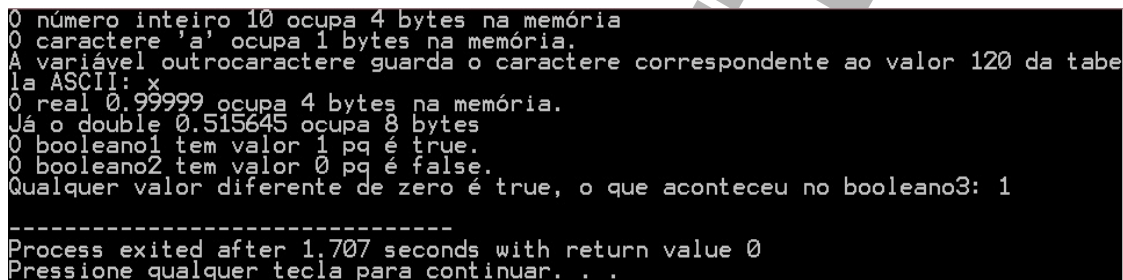
Observe que usei o comando **sizeof** em todos os couts. Esse comando é uma função, e fará mais sentido quando estudarmos funções, mas, por enquanto, entenda que ele mostra o tamanho que a variável entre parêntesis ocupa na memória. Ou seja, na linha 9, por exemplo, quando damos um “sizeof(inteiro)” ele verifica o tipo do dado e, sendo int, retorna que ele ocupa 4 bytes. Apenas a título de curiosidade, chamamos os valores dentro dos parêntesis das funções de **parâmetros**, entraremos nisso mais tarde.

Outra novidade é o **endl**. Ele tem o efeito de “quebrar a linha”, ou seja, o programa passa para uma nova linha assim que vê esse comando. Veja o programa em execução na Figura 3.10 e repare que todos os couts estão em linhas diferentes, isso se deve ao fato de ter endl no fim deles. Sem os endl todo o texto do programa ficaria em uma única linha. Faça esse algoritmo e deixe sem os endl pra ver como fica. Lembrando que todos os códigos-fonte deste manual estão no meu GitHub: **igorcaavalcante**.

Porém, observe que a linha 25 não possui endl e no programa em execução o texto encontra-se quebrado em linhas diferentes. Isso porque usei “\n” dentro no texto e aí quando o compilador encontra esse sinal ele quebra a linha.

Lembra que te falei sobre a tabela ASCII (caso não lembre, está na seção 3.2.2.1)? Tendo em vista essa correspondência entre números e caracteres, caso você coloque um número numa variável char ele considerará o valor correspondente na tabela. Ou seja, o número 120 corresponde à letra 'x', então a variável outrocaractere guarda o x.

Digo e repito: faça você mesmo os algoritmos, mude coisas, veja o que acontece caso mude um valor, uma letra, assim que se aprende a programar. Veja na Figura 3.10 esse algoritmo em execução.



```
0 número inteiro 10 ocupa 4 bytes na memória
0 caractere 'a' ocupa 1 bytes na memória.
A variável outrocaractere guarda o caractere correspondente ao valor 120 da tabela ASCII: x
0 real 0.99999 ocupa 4 bytes na memória.
Já o double 0.515645 ocupa 8 bytes
0 booleano1 tem valor 1 pq é true.
0 booleano2 tem valor 0 pq é false.
Qualquer valor diferente de zero é true, o que aconteceu no booleano3: 1

-----
Process exited after 1.707 seconds with return value 0
Pressione qualquer tecla para continuar. . .
```

Figura 3.10: Programa das variáveis rodando. Fonte: Autor

Observe que coloquei um valor de 15 casas decimais na variável double, mas só foram exibidos 6 números depois da vírgula. Isso aconteceu porque o cout é configurado por padrão para exibir apenas as 5 primeiras casas. Para mostrar todas, adicione, na linha 19, o seguinte comando:

cout.precision(15);

Onde 15 é o número de casas decimais que você quer exibir no cout.

3.2.4 Modificadores de tipo

Nós podemos fazer algumas “alterações” nesses tipos de dados de forma que fiquem ainda mais adaptados às nossas necessidades: para isso, usamos os Modificadores de Tipo.

Imagine que você precisa de uma variável que guarde o número de um dia. Não sei o seu, mas no meu calendário um mês pode ter no máximo 31 dias. Para isso, usaríamos o tipo int, mas como foi visto na tabela 3.1, o tipo inteiro pode armazenar valores de -2.147.483.648 até 2.147.483.647. Eis o X da questão: por que usar tanto espaço pra guardar apenas um número que vai de 1 à, no

máximo, 31?? Pra minimizar esse excesso, foram criados os modificadores de tipo, que diminuem ou aumentam o tamanho das variáveis para evitar excessos ou ainda aumentar a capacidade das variáveis. Observe quais são na tabela 3.2.

Modificador	Descrição
short	Diminui o tamanho da variável
long	Aumenta o tamanho da variável
unsigned	Desconsidera valores negativos
signed	Considera positivos e negativos

Tabela 3.2: Modificadores de Tipo

O **short** é usado em variáveis do tipo `int` para reduzir pela metade seu tamanho e, consequentemente, a abrangência dos valores possíveis. Já o **long** faz o oposto: dobra a capacidade das variáveis e consegue armazenar valores ainda maiores. Falando em números, o `short int` armazena de -32.768 a 32.767 e o `long` de $-9.223.372.036.854.775.808$ to $9.223.372.036.854.775.807$.

O **signed** informa que “o sinal importa” para essa variável, ou seja, ela deve armazenar valores positivos e negativos. Porém, você deve ter observado que o padrão para variáveis inteiras é aceitar tanto positivos quanto negativos, então é como se esse modificador estivesse implícito na declaração de inteiros comuns, sendo assim, é um modificador não muito utilizado. Já o operador **unsigned** ignora valores negativos e aumenta a abrangência dos números positivos, já que não diminui o espaço alocado na memória. Isso quer dizer que um “`int`” e um “`unsigned int`” ocupam os mesmos 4 bytes de memória, mas o `int` armazena valores de $-2.147.483.648$ até $2.147.483.647$ e o `unsigned int` armazena o dobro dos positivos: de 0 a $4.294.967.295$.

Mas atenção: alguns compiladores de C++ não dão suporte à algumas dessas funcionalidades e nem mostram que deu “erro”, então verifique antes se seu compilador aceita isso (a Figura 3.11 mostra uma forma de ver se o compilador aumenta ou diminui os tamanhos). Além disso, variáveis tipo `float` não podem ser modificadas.

3.2.4.1 Modificadores de tipo na prática

Veremos agora na Figura 3.11 os modificadores na prática.

O principal objetivo desse algoritmo é exemplificar a relação entre os modificadores, seus tipos e o espaço de memória alocado para essas variáveis dependendo do seu tipo, já que o tipo `int`, geralmente de 4 bytes, pode ocupar mais ou menos memória dependendo do modificador usado. Sem mais delongas, observe na Figura 3.12 o programa em execução no Windows e no Ubuntu.

Sobre essa execução: logo de cara, o compilador padrão do Dev C++ (Windows) não aumenta o tamanho do `long int`, já que ele ocupa os mesmos 4 bytes, e não 8, como deveria ser. Rodei o mesmo programa no meu Ubuntu e o resultado pode ser conferido na parte inferior da Figura 3.12.

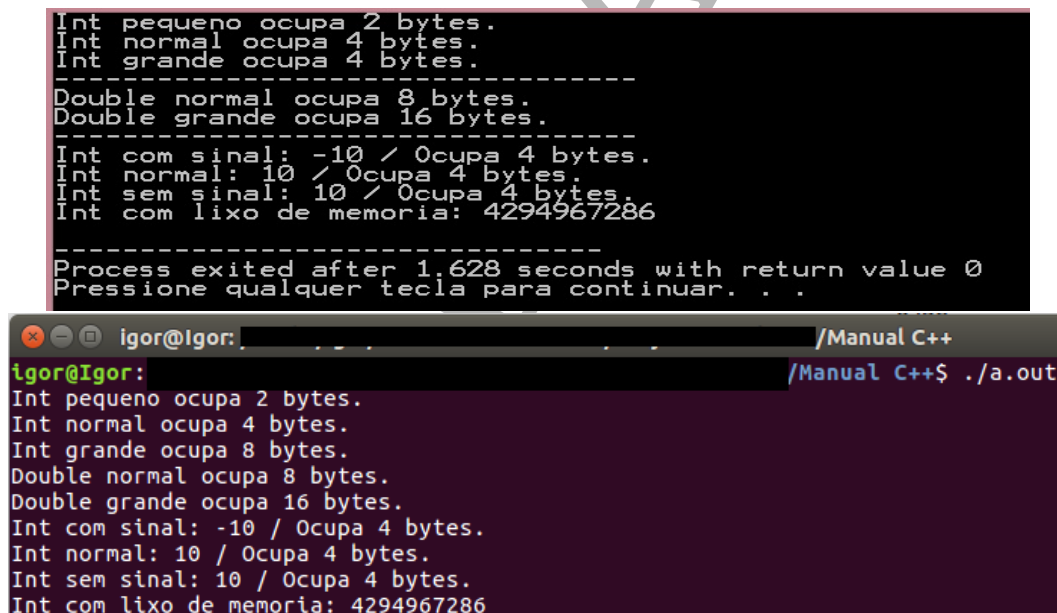
Além disso, fiz questão de colocar um número negativo numa variável com o modificador `unsigned`, ou seja, um modificador que não permite armazenar valores negativos. Quando isso acontece, o computador pega um valor aleatório que está na RAM e coloca na variável. Chamamos esse valor aleatório de **lixo de memória** por não ter sentido algum e por aparecer as vezes quando fazemos algo de errado assim como aconteceu agora.

```

modificadores.cpp
1  #include <iostream>
2  //Programa para exemplificar o uso das variáveis.
3  int main(){
4
5      using namespace std;
6
7      short int intpequeno;
8      int intnormal;
9      long int intgrande;
10     cout << "Int pequeno ocupa " << sizeof(intpequeno) << " bytes." << endl;
11     cout << "Int normal ocupa " << sizeof(intnormal) << " bytes." << endl;
12     cout << "Int grande ocupa " << sizeof(intgrande) << " bytes." << endl;
13
14     double doublenormal;
15     long double doublegrande;
16     cout << "Double normal ocupa " << sizeof(doublenormal) << " bytes." << endl;
17     cout << "Double grande ocupa " << sizeof(doublegrande) << " bytes." << endl;
18
19     signed int comsinal = -10;
20     int intnormal2 = 10;
21     unsigned int semsinal = 10;
22     unsigned int intcomlixo = -10;
23     cout << "Int com sinal: " << comsinal << " / Ocupa " << sizeof(comsinal) << " bytes." << endl;
24     cout << "Int normal: " << intnormal2 << " / Ocupa " << sizeof(intnormal2) << " bytes." << endl;
25     cout << "Int sem sinal: " << semsinal << " / Ocupa " << sizeof(semsinal) << " bytes." << endl;
26     cout << "Int com lixo de memoria: " << intcomlixo << endl;
27
28     return 0;
29 }

```

Figura 3.11: Programa para exemplificar o uso dos modificadores de tipo. Fonte: Autor



```

Int pequeno ocupa 2 bytes.
Int normal ocupa 4 bytes.
Int grande ocupa 4 bytes.
-----
Double normal ocupa 8 bytes.
Double grande ocupa 16 bytes.
-----
Int com sinal: -10 / Ocupa 4 bytes.
Int normal: 10 / Ocupa 4 bytes.
Int sem sinal: 10 / Ocupa 4 bytes.
Int com lixo de memoria: 4294967286
-----
Process exited after 1.628 seconds with return value 0
Pressione qualquer tecla para continuar. . .

igor@Igor: /Manual C++
igor@Igor: /Manual C++$ ./a.out
Int pequeno ocupa 2 bytes.
Int normal ocupa 4 bytes.
Int grande ocupa 8 bytes.
Double normal ocupa 8 bytes.
Double grande ocupa 16 bytes.
Int com sinal: -10 / Ocupa 4 bytes.
Int normal: 10 / Ocupa 4 bytes.
Int sem sinal: 10 / Ocupa 4 bytes.
Int com lixo de memoria: 4294967286

```

Figura 3.12: Programa dos modificadores executando, respectivamente, no Windows e no Ubuntu. Fonte: Autor

3.2.5 Alguns cuidados necessários

Essa rápida seção mostra alguns cuidados que você precisa ter ao trabalhar com variáveis:

- Variáveis podem ser nomeadas com: números, letras e sublinhados (_). **Não podem** ser iniciadas com números e não podem conter caracteres especiais como asteriscos (*), cerquilhas (#), e comerciais (&), acentos, etc. Único sinal que pode ser usado em qualquer lugar da variável é o _, mas que deve ser evitado no início das variáveis porque podem gerar conflitos com comandos internos do C++.
- Elas devem ser nomeadas da forma mais didática possível, de forma que você consiga entender seu propósito só de ler seu nome. Ou seja, uma variável que armazena uma média pode ser chamada de “media” (não use acentos!!!).
- Elas podem ser declaradas em qualquer lugar do programa, só tenha cuidado para declará-las antes de fazer alguma operação com ela.
- A linguagem é baseada no inglês, então números reais tem pontos separando casas decimais, não vírgula como usamos.
- Por fim, **o mais importante:** C++ é uma linguagem **case sensitive**, ou seja, ela **faz a diferenciação entre minúsculas e maiúsculas**. Traduzindo: “a” e “A” são letras totalmente diferentes.

Observe na Figura 3.13 como essas dicas importam na prática.

3.2.6 Lendo dados

Até agora nossos programas não tem interação com o usuário, ou seja, os dados trabalhados nele já são definidos no algoritmo, mas obviamente programas são projetados para interagir com usuários, pra que possam inserir dados e fazer algo com eles. Sendo assim, precisamos captar os dados inseridos através do teclado. Confira na Figura 3.14 um programa que soma dois números informados pelo usuário.

A novidade agora é o comando **cin** (**C**onsole **I**nput). Quando o programa chega nessa linha, fica aguardando o usuário digitar algo, continuando apenas quando a tecla enter é apertada. Além disso, observe que os sinais não são de menor, e sim de maior que. Isso porque as operações são inversas: o cout é um comando de **saída** de dados e o cin de **entrada**.

3.2.7 Operadores aritméticos

Até agora nós simplesmente jogamos valores em variáveis, mas na prática nós precisaremos fazer várias operações com esses valores: multiplicações, divisões, somas, etc. Alguns, como a soma, são intuitivos pra gente, mas outros, como o do módulo, nem tanto.

Considere as seguintes variáveis com seus respectivos valores para usarmos nos exemplos:

```

1  #include <iostream>
2
3  /*Este programa serve para exemplificar os
4  cuidados que devemos ter na com as variáveis*/
5
6  int main() {
7      using namespace std;
8
9      //Quatro variáveis TOTALMENTE DIFERENTES:
10     int aa, aA, Aa, AA;
11
12     int kxz; //nome nada descritivo
13     float media_aritimetica; //nome bem descritivo
14
15     //Podemos usar _ e números:
16     double _media, _media2, me2dia;
17
18     //mas essas DÃO ERRO!!:
19     float #soma, !numero1, 12media;
20
21     //Isso vai dar ERRO:
22     cout << "A media e: " << media;
23     float media = 6.5;
24     /*Pois tentamos usar a variável ANTES
25     dela ter sido declarada!*/
26
27     return 0;
28 }

```

Figura 3.13: Cuidados no uso das variáveis. Fonte: Autor

```

1  #include <iostream>
2
3  /*Programa que soma dois números*/
4
5  int main() {
6      using namespace std;
7
8      int numero1, numero2, soma;
9
10     cout << "Digite o primeiro numero: ";
11     cin >> numero1;
12
13     cout << "Digite o segundo numero: ";
14     cin >> numero2;
15
16     soma = numero1 + numero2;
17
18     cout << "A soma " << numero1 << "+" << numero2 << " = " << soma;
19
20     return 0;
21 }

```

Figura 3.14: Lendo e somando 2 números. Fonte: Autor

```
int a = 2;
```

```
int b = 4;
```

Começando pelo mais básico, podemos fazer as 4 principais operações da matemática:

```
cout << a + b; //exibe a soma
cout << a - b; //exibe a subtração
cout << a * b; //exibe a multiplicação
cout << b / a; //exibe a divisão
```

Um detalhe: a divisão de a por b não exibiria o resultado esperado, pois o resultado não é um número inteiro. Para isso, poderíamos usar floats ou fazer outras gambiarras que podem ser abordadas depois.

Além disso, temos o operador do módulo, que retorna o resto da divisão entre dois inteiros, que é representado assim:

```
int x = b % a; // x recebe o resto da divisão de b por a
```

Considerando que o valor de b é 4 e a é 2, x vai receber o valor 0, já que a divisão não tem resto.

O resultado do módulo é útil, por exemplo, para definir se um número é par ou ímpar. Se o resto da divisão do número por 2 resultar em 0, o número é par. Se resultar 1, é ímpar. Pode fazer no papel aí com qualquer número e vai ver que dá certo. Veja na seção 4.1.1.3 um exemplo de tal programa.

É corriqueiro adicionarmos ou retirarmos uma unidade à uma variável. Por exemplo: caso queiramos guardar a quantidade de números digitados em um programa, podemos usar uma variável inteira que ganha uma unidade toda vez que um número é digitado. Intuitivamente, você já deve estar pensando que basta digitar

```
x = x + 1;
```

e tá tudo certo (x recebe o valor de x mais 1 unidade, ou seja, se x valesse 5, agora valeria $5 + 1$). Realmente tá, mas a linguagem facilita pra gente. Essa atribuição poderia ser escrita da seguinte forma:

```
x++; //mesma coisa que "x = x + 1;"
```

O mesmo aconteceria caso quisesse tirar uma unidade:

```
x--; //mesma coisa que "x = x - 1;"
```

Essa é apenas uma forma mais simples de escrever algo e deixar mais visível que estamos acrescentando uma unidade. Com a prática fica até mais intuitivo escrever assim. Chamamos isso de operadores de incremento pós-fixados.

Esta linguagem permite, ainda, incrementos/decrementos pré-fixados, que escrevemos da seguinte forma:

```
-x; ++x;
```

```

1  #include <iostream>
2  /*Programa para exemplificar a diferença entre
3  operadores pre e pos fixados*/
4
5  int main() {
6      using namespace std;
7
8      int A=1, B=2; //Declarando as variáveis
9
10     A = B++; //NESTA LINHA, colocamos o valor de B (2) em A e DEPOIS incrementamos 1 em B
11     cout << "A:" << A << " B:" << B << endl; //NESTA LINHA, B vale 3 pq ganhou 1 na linha anterior e A vale 2
12
13     A = ++B; //NESTA LINHA, incrementamos 1 em B (agora vai valer 4 pq tinha 3 antes) e DEPOIS colocamos o valor (4) em A
14     cout << "A:" << A << " B:" << B << endl; //NESTA LINHA, A e B tem o mesmo valor (4)
15
16     return 0;
17 }

```

Figura 3.15: Programa que diferencia operadores pré e pós fixados. Fonte: Autor

O resultado final disso é o mesmo: tiramos ou acrescentamos uma unidade, a diferença é apenas QUANDO isso acontece. Veja um exemplo na Figura 3.15.

Resumindo: os pré-fixados fazem o incremento e depois a atribuição e os pós realizam primeiro a atribuição e depois o incremento. Outra forma de explicar: os pré-fixados realizam o incremento “na mesma linha”, e os pós-fixados “só valem” na próxima linha.

Ainda nessa ideia de escrever atribuições de forma mais simples, podemos fazer as seguintes operações:

```

a += b; //mesmo que "a = a + b;"
a -= b; //mesmo que "a = a - b;"
a *= b; //mesmo que "a = a * b;"
a /= b; //mesmo que "a = a / b;"
a %= b; //mesmo que "a = a % b;"

```

Deixarei sob sua responsabilidade colocar isso em prática e ver como funciona.

Ainda existem os operadores comparativos, que serão abordados na seção 4.1.1.1.

3.2.8 Chars: letras e palavras (EM CONSTRUÇÃO)

O tipo char é o tipo responsável por guardar caracteres num geral: letras, números, símbolos, etc. Na seção 3.2.2.1 eu citei a tabela ASCII, que faz a correspondência entre os caracteres e números, sugiro que de uma olhada na seção e faça uma pesquisa pra saber mais. Mas há um detalhe: uma variável char guarda apenas UM caractere (existem formas de guardar mais, mas por enquanto vamos focar na parte mais simples). A declaração segue o modelo dos tipos anteriores. Veja na Figura 3.16 um exemplo super simples de programa com o tipo char. Faça o mesmo programa e fique fazendo testes nele, explorando TODAS as possibilidades.

Se você testou direitinho e explorou realmente todas as possibilidades, vai ter encontrado alguns probleminhas nesse algoritmo.


```

1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6
7      char resposta;
8      int a, b;
9
10     cout << "Voce quer fazer uma soma? Digite: S - SIM | N - NAO -> ";
11     cin >> resposta;
12
13     if(resposta == 'S'){
14         cout << "Digite o primeiro valor: ";
15         cin >> a;
16         cout << "Digite o primeiro valor: ";
17         cin >> b;
18         cout << a << " + " << b << " = " << a+b << endl;
19     }
20
21     else{
22         cout << "Entao pq executou esse programa???" << endl;
23     }
24 }

```

Figura 3.16: Você quer somar dois números, @?. Fonte: Autor

O primeiro, e mais visível é o seguinte: se você digitar um “s” ele vai achar que você digitou um “N”. Se você não tinha percebido, rode de novo e constate isso. Isso se deve ao fato de C++ ser *case sensitive*. Como foi citado na seção 3.2.5, ser case sensitive significa que “S” e “s” são letras TOTALMENTE diferentes, já que a linguagem faz essa diferenciação. Volte lá na tabela ASCII e veja que o número relacionado à letra “S” e o relacionado a “s” são diferentes. Nós, como programadores, devemos pensar em todas as possibilidades, já que nem sempre o usuário vai perceber uma diferença sutil assim. Logo, pense um pouco sobre como poderíamos fazer pra contornar esse pequeno problema. Pensou? Bem, minha sugestão é a seguinte: C++ possui uma função que transforma qualquer letra em maiúscula, sendo assim, é só converter a letra inserida pra maiúscula e fazer a comparação com o “S”. Caso ela já esteja maiúscula a função não irá alterar e fazer a comparação do mesmo jeito.

Outro probleminha: roda o programa e digita um “G” pra ver o que acontece. Ele considera que você escreveu “N”, já que qualquer valor diferente de “S” cai no else e é considerado “N”. Ou seja, eu não coloquei um if pra verificar se o usuário digitou “N”, apenas verifiquei o “S” e qualquer outro valor vai direto pro else. A Figura 3.17 mostra o mesmo algoritmo com as alterações necessárias.

Dito isso, vamos melhorar um pouco a brincadeira: variáveis do tipo char podem armazenar também palavras, que nada mais são do que um conjunto de caracteres, certo? Palavras, no universo da programação, são chamadas de **strings**.

*****IMAGEM AQUI*****

```

1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6
7      char resposta;
8      int a, b;
9
10     cout << "Voce quer fazer uma soma? Digite: S - SIM | N - NAO -> ";
11     cin >> resposta;
12
13     if(toupper(resposta) == 'S'){
14         cout << "Digite o primeiro valor: ";
15         cin >> a;
16         cout << "Digite o primeiro valor: ";
17         cin >> b;
18         cout << a << " + " << b << " = " << a+b << endl;
19     }
20
21     else{
22         if(toupper(resposta) == 'N')
23             cout << "Entao pq executou esse programa??" << endl;
24         else
25             cout << "Nao ha opcao correspondente a letra digitada, idiota." << endl;
26     }
27 }

```

Figura 3.17: Você quer somar dois números, @? (versão melhorada). Fonte: Autor

3.3 Constantes

Constantes são como as variáveis, a diferença é que são constantes (risos). Digo que são como variáveis por que constantes e variáveis compartilham duas características: ambas possuem um nome e um valor. A diferença é que o valor da constante não pode ser alterado: uma vez definido será assim até o fim da execução.

O melhor exemplo é a constante **pi** que trazemos da matemática. Não sei no seu, mas no meu mundo o valor de pi NUNCA muda. Sendo assim, pi é uma constante e pode ser definida em programação como tal.

Imagine um programa que faz vários cálculos que envolvem o valor de pi. Pra não ter que ficar colocando o valor de pi (aproximadamente 3.14159265359) toda vez que formos usá-lo, podemos simplesmente definir que $\pi = 3.14159265359$ e usar apenas “pi” no programa inteiro para se referir a esse valor. Além disso, um nome é mais expressivo do que um número. Então, se você usar pi ao invés de 3.1415926535, seu cálculo fica mais limpo e fácil de entender. Construa seus programas de forma que manutenções sejam mais fáceis, seu futuro agradece!

Veja as constantes na prática na Figura 3.18.

```
1  #include <iostream>
2
3  #define pi 3.14159
4  #define meunome "Iguinho"
5
6  int main(){
7      using namespace std;
8
9      cout << "O valor de pi eh " << pi << endl;
10
11     cout << "pi x 2 = " << pi * 2 << endl;
12
13     cout << "Meu nome eh " << meunome;
14
15     return 0;
16 }
```

Figura 3.18: Constantes na prática. Fonte: Autor

Capítulo 4

Estruturas de seleção e de repetição

Estruturas de seleção e de repetição alteram o fluxo do algoritmo. Até aqui, nossos programas vão sendo executados de cima pra baixo linha por linha, mas essas estruturas fazem com que linhas sejam “puladas” ou repetidas. Eu sei, o que você acabou de ler pode não fazer muito sentido pra você agora, mas acredito que nas próximas páginas isso ficará óbvio de tão claro.

4.1 Tomando decisões: estruturas de seleção

Estruturas de seleção são usadas para definir um fluxo, um caminho que um algoritmo deve percorrer geralmente de acordo com informações externas, que podem ser um valor fornecido pelo usuário, informações contidas em arquivos, etc. É como se criássemos bifurcações: caminhos diferentes de acordo com determinados fatores. Ainda não faz sentido, né? Calma.

4.1.1 *if...else...*

Imagine um programa de caixa automático, em especial a parte do saque. Quando um usuário vai sacar determinado valor, o caixa precisa verificar se ele tem dinheiro suficiente pra isso, certo? Então, **SE** o saldo do usuário for **maior ou igual** ao valor solicitado, **então** ele pode fazer o saque. **SE NÃO**, seu saldo é insuficiente. Percebeu que isso é um algoritmo? Não? Veja o pseudocódigo abaixo considerando que o usuário tem R\$100 na conta e quer sacar R\$50:

```
saldo = 100
valor_solicitado = 50

SE valor_solicitado ≤ saldo, ENTÃO:
    liberar o saque!
SE NÃO:
    saldo insuficiente!
```

Observe que o programa, ao chegar num comando de decisão, ele pode seguir dois “caminhos”: o primeiro é se a condição for satisfeita, o segundo é se a condição não for. Decidir o “caminho” que um programa deve seguir baseado em uma comparação de valores é o que chamamos de **tomada**

de decisão ou desvio condicional. Observe a Figura 4.1 para ver graficamente como isso funciona.

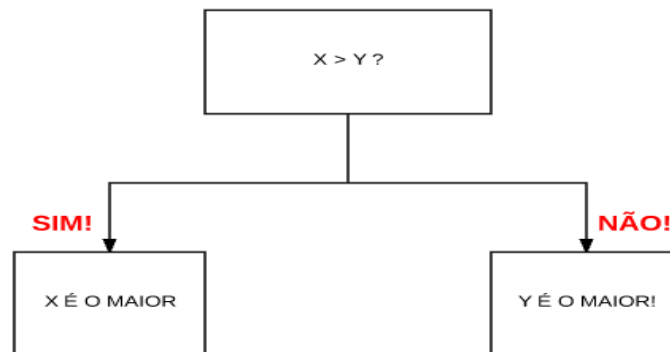


Figura 4.1: Fluxograma para exemplificar os “caminhos” que um programa pode seguir após uma tomada de decisão. Fonte: Autor

Pegando o exemplo do fluxograma acima, veja na Figura 4.2 como poderíamos implementar esse algoritmo em C++.

```
1  #include <iostream>
2
3  /*Programa que define qual o maior
4  de dois números.*/
5
6  int main() {
7      using namespace std;
8
9      int x, y;
10
11     cout << "Digite o X: ";
12     cin >> x;
13
14     cout << "Digite o Y: ";
15     cin >> y;
16
17     if(x > y){
18         cout << "X é maior!";
19     }
20
21     else{
22         cout << "Y é maior!";
23     }
24
25     return 0;
26 }
```

Figura 4.2: Programa que define o maior entre dois números. Fonte: Autor

Falando de forma mais técnica, ao chegar em uma estrutura de seleção, o programa **avalia** a operação (mais tarde veremos que podem existir várias operações em um único if) que está entre parêntesis. Essa avaliação significa fazer determinada comparação entre valores e gerar um resultado: *true* ou *false* (verdadeiro ou falso). Se o resultado for *true*, ou seja, se a condição for

satisfeita (exemplo: $4 > 3$ é true, $3 < 4$ também, mas $4 < 3$ é false), o programa executa o que está dentro dos parêntesis. Se não (*else*) ele vai executar o que está dentro do *else* ou, caso não tenha um *else*, não faz nada e segue em frente.

Dica simples: quando estiver lendo isso, ao invés de “if...else” leia como “se, se não”, talvez essa simples mudança faça com que as coisas sejam mais claras na sua cabeça, o que é importante quando tratamos de programação.

Mas pense um pouco: o que aconteceria se o usuário digitasse dois números iguais nesse programa? Tente resolver esse problema! Dica: você pode colocar vários desvios condicionais dentro de um desvio condicional. Ou seja, você pode colocar um *if...else* dentro de outro *if...else*.

Um detalhe rápido: um *else* tem que ter um *if*, mas um *if* não precisa ter um *else*. Em outras palavras: só há um “se não” se houver um “se”.

4.1.1.1 Operadores comparativos (relacionais)

Os operadores comparativos -pasmem!- servem para comparar valores: se são maiores, menores, iguais, etc. Sem querer querendo vocês intuitivamente acabaram entendendo como definir o menor/maior de dois números, mas podemos fazer outras comparações, geralmente dentro de estruturas de seleção:

```
1  #include <iostream>
2
3  /*Programa para exemplificar o uso de operadores de comparação*/
4
5  int main() {
6      using namespace std;
7
8      int A, B; //Declarando as variáveis
9
10     //Lendo os valores das variáveis:
11     cout << "Digite um valor para A: ";
12     cin >> A;
13     cout << "Digite um valor para B: ";
14     cin >> B;
15
16     //Podemos realizar as seguintes comparações:
17
18     if(A > B)
19         cout << "A é maior que B!" << endl;
20     if(A < B)
21         cout << "A é menor que B!" << endl;
22
23     if(A <= B)
24         cout << "A é menor ou igual a B!" << endl;
25     if(A >= B)
26         cout << "A é maior ou igual a B!" << endl;
27
28     if(A == B)
29         cout << "A é igual a B!" << endl;
30     if(A != B)
31         cout << "A é diferente de B!" << endl;
32
33     return 0;
34 }
```

Figura 4.3: Operadores comparativos. Fonte: Autor.

De cara: como os ifs **só têm um comando** (eu disse comando, não linha, podemos ter mais de um comando em uma linha) eu pude escrevê-los sem chaves. Veja na seção ?? o que você pode ou não fazer. Recomendo usar sempre chaves, mas fiz isso pra deixar a imagem menor.

Acredito que seja simples demais entender o intuito de cada comparação dessas, então vou complicar um pouco as coisas: vamos colocar mais de uma comparação dentro de um único if. Para estabelecer relações entre elas usaremos os **operadores lógicos**.

4.1.1.2 Operadores Lógicos

Operadores lógicos servem para estabelecer relações entre comparações. Por exemplo: caso nosso programa queira um número entre 100 e 200, nós precisamos verificar se ele é **maior que 100 E menor que 200**, concorda? Podemos escrever, agora mais perto da programação, da seguinte forma:

(numero > 100) **E** (numero < 200)

Ou seja, as duas condições necessariamente precisam ser verdadeiras (true). Sendo assim, podemos dizer que o “E” é um operador lógico, pois estabelece uma relação entre duas comparações. Além do “E”, temos também o “OU” e a negação. Veja abaixo:

Tabela 4.1: Operadores lógicos

Operador	Símbolo	Exemplo	É true se:
OR (OU)		if((expressao1) (expressao2))	Pelo menos uma for verdadeira
AND (E)	&&	if((expressao1) && (expressao2))	As duas forem verdadeiras
NOT (NÃO)	!	if(!(expressao))	O resultado da expressão for false

Veja na Figura 4.4 um exemplo prático. É um exemplo um pouco mais complexo de se entender, faça isso você mesmo e fique alterando pra ver como o programa se comporta. Observe que a linha 26 tem condições dentro de condições. Agora lembre das expressões matemáticas. Por exemplo, vamos resolver $[(2+4)+5]+10$:

$$[(2+4)+5]+10 = [6+5]+10 = 11+10 = 21$$

Na programação acontece algo parecido: o programa começa avaliando o que está mais interno nos parêntesis e vai avaliando o resto.

4.1.1.3 Exemplo prático: par ou ímpar?

Um programa que determina se um número é par ou ímpar é um ótimo exemplo de como usar condições e o operador de módulo citado na seção 3.2.7. Veja na Figura 4.1.1.3.

```

1  #include <iostream>
2  /*Programa para exemplificar o uso dos operadores lógicos*/
3
4  int main() {
5      using namespace std;
6      int A=1, B=2, C=1, D=5; //Declarando as variáveis
7
8      if((A<B) || (A==B)){
9          //Entra aqui, pois pelo menos uma comparação é verdadeira!
10     }
11     if((A>B) || (A!=C)){
12         //NÃO entra aqui, pois nenhuma comparação é verdadeira!
13     }
14     if((A<B) || (A!=B)){
15         //Entra aqui, pois as duas comparações são verdadeiras!
16     }
17     if((A<B) && (A!=B)){
18         //Entra aqui, pois as duas comparações são verdadeiras!
19     }
20     if((A<B) && (A==B)){
21         //NÃO entra aqui, pois uma das duas é falsa!
22     }
23     if((A<B) && (A==C) && (A!=D) && (B<A)){
24         //NÃO entra aqui, pois uma delas é falsa!
25     }
26     if((A<B) && ((A<B) || (D>B)) && (A!=D) && ((A<B) || (B<C))){
27         //Entra aqui, pois as TODAS comparações são verdadeiras!
28     }
29     return 0;
30 }

```

Figura 4.4: Operadores lógicos. Fonte: Autor.

```

1  #include <iostream>
2  /*Par ou ímpar?*/
3
4  int main() {
5      using namespace std;
6
7      int numero;
8
9      cout << "Digite um numero: ";
10     cin >> numero;
11
12     if(numero%2==0){
13         cout << "O numero " << numero << " eh par.";
14     }else{
15         cout << "O numero " << numero << " eh impar.";
16     }
17
18     return 0;
19 }

```

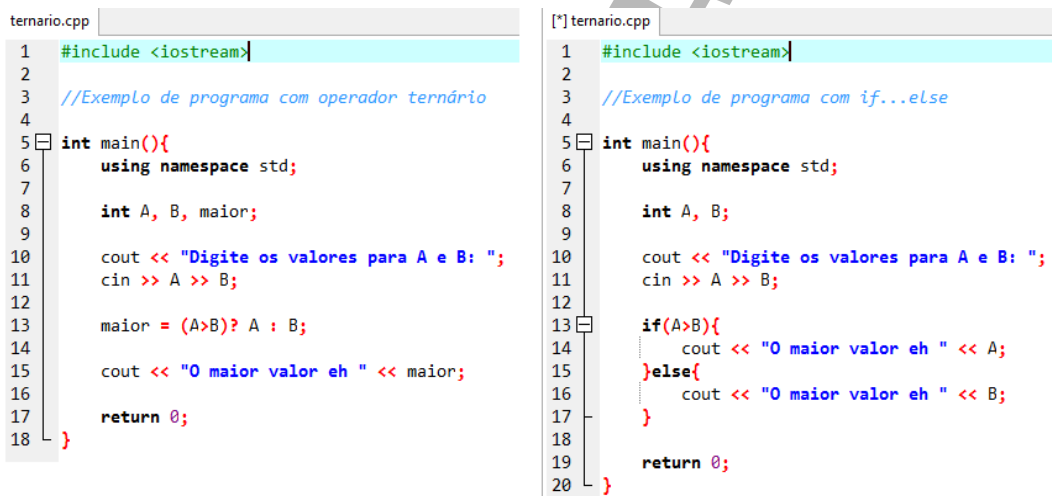
Figura 4.5: Par ou ímpar?. Fonte: Autor.

4.1.1.4 Operador ternário

Um operador ternário é tipo um *if...else*, só que mais simples ainda, a diferença é que ele não executa um bloco de código, mas apenas uma atribuição. Funciona da seguinte forma: você define uma variável que poderá assumir dois valores dependendo de uma condição. Caso a condição seja satisfeita (resultado da condição for *true*) a variável receberá o primeiro valor, se não, receberá o segundo valor. A sintaxe é a seguinte:

$$\text{variável} = (\text{condição}) ? \text{valorSeCondicaoForTrue} : \text{valorSeCondicaoForFalse};$$

Operador ternário é uma solução limpa, prática e elegante pra uma operação que tomaria algumas linhas se fosse feita com *if...else*. Observe que nem todo *if...else* pode ser escrito com operadores ternários, já que o último trata apenas de uma atribuição e o *if...else* pode conter mais comandos em sua estrutura. Veja na Figura 4.6 o mesmo programa feito com *if...else* e com operador ternário.



The figure displays two code snippets side-by-side, comparing the implementation of finding the maximum of two numbers using the ternary operator versus an if-else statement.

Left Snippet (ternario.cpp):

```
1 #include <iostream>
2
3 //Exemplo de programa com operador ternário
4
5 int main(){
6     using namespace std;
7
8     int A, B, maior;
9
10    cout << "Digite os valores para A e B: ";
11    cin >> A >> B;
12
13    maior = (A>B)? A : B;
14
15    cout << "O maior valor eh " << maior;
16
17    return 0;
18 }
```

Right Snippet (ternario.cpp):

```
1 #include <iostream>
2
3 //Exemplo de programa com if...else
4
5 int main(){
6     using namespace std;
7
8     int A, B;
9
10    cout << "Digite os valores para A e B: ";
11    cin >> A >> B;
12
13    if(A>B){
14        cout << "O maior valor eh " << A;
15    }else{
16        cout << "O maior valor eh " << B;
17    }
18
19    return 0;
20 }
```

Figura 4.6: Maior de dois números com operador ternário e com *if...else*. Fonte: Autor

4.1.2 *switch...case...*

O *switch* serve como uma sequência de *if...else*, só que de forma mais simples e legível. A única diferença é que o *switch* não permite fazer operações lógicas. Não entendeu? Ainda na ideia do caixa automático, veja o pseudocódigo abaixo de um exemplo de menu para ter uma ideia melhor:

Digite:
1 para saldo
2 para saque
3 para depósito

Caso digite 1: /*Mostrar o saldo*/
Caso digite 2: /*Realizar o saque*/
Caso digite 3: /*Realizar o deposito*/

Agora, estude o mesmo algoritmo em C++ na Figura 4.7.

```
1  #include <iostream>
2
3  /*Programa que simula um caixa automático para
4  exemplificar o uso do switch...case...*/
5
6  int main(){
7      using namespace std;
8      float saldo = 100; //Defini um saldo de R$100
9      float valor; //Usaremos para operações de saque/deposito
10     int opcao_selecionada; //Usaremos para guardar a opção desejada
11
12     cout << "*****Caixa automatico*****" << endl; //Só pra deixar bonito
13     cout << "\n0 que deseja fazer?: \n1-Saldo \n2-Saque \n3-Deposito \n>>";
14     cin >> opcao_selecionada;
15
16     switch(opcao_selecionada){
17         case 1: cout << "Seu saldo eh: " << saldo; break; //dá pra fazer assim em uma linha
18
19         //mas assim fica mais legível:
20         case 2:
21             cout << "Qual valor a ser sacado? >> R$";
22             cin >> valor;
23             if(valor <= saldo){
24                 saldo -= valor; // <-Isso é a mesma coisa que isto-> saque = saque - valor; !!!
25                 cout << "Saque efetuado com sucesso! Novo saldo: " << saldo;
26             }else{
27                 cout << "Saldo insuficiente!";
28             }
29             break;
30
31         default: cout << "O numero digitado nao corresponde a nenhuma opcao";
32     }
33     return 0;
34 }
```

Figura 4.7: Simulação de um caixa automático para exemplificar o uso do *switch*. Fonte: Autor.

Obviamente esse é um exemplo bem rudimentar de um caixa, a começar pelo saldo inicial fixado em R\$100, mas dá pra ter uma ideia de como os caixas funcionam.

Primeiro, pedi pra o usuário digitar o número correspondente à opção desejada. Na linha 16 usei o *switch* passando nos parêntesis a variável que guardou o número da opção. Depois disso, o programa faz a comparação do número digitado com os “cases” que definimos. Leia a linha 17 assim: “caso (case) o usuário tenha digitado 1, mostre o saldo”.

Logo, internamente é como se o programa fizesse um *if...else* assim: `if(opcaoselecionada == 1)`, só que escrito de uma forma mais simples e legível (imagine esse menu com 100 opções e a gente ter que fazer 100 *if...else* a bagunça que ficaria).

Sobre a linha 31: imagine que o usuário por algum descuido (ou burrice mesmo) digite um número que não possui opção correspondente. É pra esse caso que serve o **default**: caso não tenha entrado em nenhum case, entra no default.

Observe que os cases, ao contrário do *if...else*, não possuem chaves para delimitar seu conteúdo, e isso está diretamente ligado ao comando **break** que tem no final das instruções dos cases. Para entender o porquê do *break*, rode o programa da Figura 4.8.

```
1  #include <iostream>
2
3  /*Programa que justifica o uso do break*/
4
5  int main(){
6      using namespace std;
7
8      int numero;
9
10     cout << "Digite um numero: ";
11     cin >> numero;
12
13     switch(numero){
14         case 1: cout << "Vc digitou o numero 1." << endl;
15         case 2: cout << "Vc digitou o numero 2." << endl;
16         case 3: cout << "Vc digitou o numero 3." << endl;
17         case 4: cout << "Vc digitou o numero 4." << endl;
18         case 5: cout << "Vc digitou o numero 5." << endl;
19
20         default: cout << "Vc digitou um numero maior que 5 ou menor que 1.";
21     }
22     return 0;
23 }
```

Figura 4.8: Porque usar o *break*. Fonte: Autor

Primeiro, digite o número 1. Depois, digite o 3. Estranho né? Por algum motivo, quando o programa “entra” em algum case e não encontra um *break* ele sai executando todos os outros posteriores. O *break* indica o fim de um case, e é por isso que não precisamos usar chaves para limitá-lo.

Você deve ter notado (espero que tenha notado) que eu não fiz o terceiro case que é o do depósito. Esse fica por sua conta. ;)

Podemos usar o *switch* com chars também, a melhor forma de exemplificar isso é com uma calculadora. Veja na Figura 4.9 e vá mudando uma coisa ou outra pra ver o que acontece.

4.1.3 Como usar as chaves

Agora que estamos trabalhando mais com blocos de códigos, saber como usar as chaves é importante. Veja na Figura 4.10 como usá-las.

```

1  #include <iostream>
2
3  /*Calculadora com switch case*/
4
5  int main(){
6      using namespace std;
7
8      char operacao; // a operação desejada
9      float n1, n2; // os dois números
10
11     cout << "Digite um numero: > ";
12     cin >> n1;
13     cout << "Digite outro numero > ";
14     cin >> n2;
15
16     cout << "Digite a operacao: > ";
17     cin >> operacao;
18
19     switch(operacao){
20         case '/': cout << n1 << "/" << n2 << "=" << n1/n2 << endl; break;
21         case '+': cout << n1 << "+" << n2 << "=" << n1+n2 << endl; break;
22         case '-': cout << n1 << "-" << n2 << "=" << n1-n2 << endl; break;
23         case '*': cout << n1 << "*" << n2 << "=" << n1*n2 << endl; break;
24
25         default: cout << "Opcao invalida, seu burro!";
26     }
27     return 0;
28 }

```

Figura 4.9: Calculadora com *switch*. Fonte: Autor

Apesar de estarem todas corretas, as duas primeiras formas são mais aconselháveis por serem mais claras de se entender principalmente em grandes programas.

4.2 *Loops*: estruturas de repetição

Você acha útil repetir um pedaço de programa? De cara, achei que seria inútil. Mas, pensando direitinho, isso é uma das coisas mais importantes num programa! Imagine ter que escrever um programa que lê 1000 dados. Declarar 1000 variáveis e, o pior, escrever 1000 cins seria um pesadelo, certo? Com as estruturas de repetição, podemos escrever apenas uma vez e o programa repete tudo pra gente!

Vou usar uma abordagem prática pra explicar isso: vamos fazer um programa que exibe os 50 primeiros números. Sem repetições nós precisaríamos escrever 50 couts com os números de 1 a 50, certo? Mas, com repetições, nós precisamos de apenas 1 cout, e o que vai variar apenas é o valor de uma variável, que, no início, vale 1 e, a cada repetição, mostra seu valor e é acrescentada outra unidade. Confuso? Talvez no início, mas veja este pseudoalgoritmo:

```

1  #include <iostream>
2
3  /*Como usar as chaves*/
4
5  int main() {
6      using namespace std;
7
8      /*!TODAS! AS FORMAS ABAIXO ESTÃO CORRETAS!*/
9
10     if(x > y){
11         cout << "X é maior!";
12     }
13
14     else
15     {
16         cout << "Y é maior!";
17     }
18
19     if(x > y){cout << "X é maior!";}
20
21     /*Caso seu bloco só tenha um comando, você
22     pode deixar sem chaves (mas não é muito
23     indicado fazer isso...):*/
24
25     if(x < y) cout << "Y é menor!";
26
27     if(y > x)
28         cout << "Y é maior!";
29
30     return 0;
31 }

```

Figura 4.10: Como usar as chaves. Fonte: Autor

```

x = 1
repita isto {
    mostre o valor de x
    acrescente 1 a x
}
enquanto x for menor ou igual a 50

```

X começa valendo 1, é mostrado seu valor e acrescentado mais uma unidade. Depois de acrescentar, ele volta pro início do “bloco” definido pelas chaves e roda de novo esse “bloco”. Exibe-se o valor de X de novo (agora vale 2) e acrescenta-se outra unidade (agora vale 3). E assim segue o ciclo até que o x tenha valor 50. É assim que mostramos de forma fácil os valores de 1 até 50.

Veja na Figura 4.11 o algoritmo anterior em C++. Acho importante que você “leia” o algoritmo em português por que acredito que faz mais sentido na nossa cabeça, por isso comentei a “tradução” em cada linha.

Uma execução desse “bloco” (entenda por bloco tudo que está dentro das chaves da estrutura de repetição) é comumente chamada de *loop*, *laço* ou *iteração*, todos sinônimos que remetem à ideia de algo que vai e volta, um ciclo, uma volta. Encarar assim é interessante pois clareia a ideia de que algoritmo vai “descendo” e depois “volta” pra um bloco ser executado de novo. Mas pense comigo: se ele fica repetindo um pedaço do código como fazer pra parar de repetir e executar o resto do programa? É pensando nisso que temos que definir uma **condição** para a estrutura de repetição, é ela que dirá quando o programa deve parar de repetir aquele pedaço de código.

```

1  #include <iostream>
2
3  //programa que mostra os valores de 1 até 50
4
5  using namespace std;
6
7  int main(){
8      int x = 1;
9
10     do{                                //faça
11
12         cout << x << endl; //isso aqui
13         x++;                //isso aqui
14
15     }while (x<=50);          //enquanto x for menor ou igual a 50
16
17     return 0;
18 }

```

Figura 4.11: Mostrando os números de 1 até 50. Fonte: Autor

Além da condição, o conceito de **contador** também é importante, pois geralmente ele nos auxilia no controle da quantidade de iterações que uma estrutura deve ter. No exemplo da Figura 4.11 o `x`, além de ser responsável por variar de valor para serem exibidos os números de 1 a 50, teve também o papel de contador, já que foi ele quem definiu quantas vezes aquela estrutura seria repetida. A Figura 4.14 mostra um exemplo em que foi necessário usar um contador exclusivamente pra definir quantos *loops* seriam executados. Mas, ao contrário da condição, o contador não é um elemento essencial: a Figura ?? mostra um exemplo que não usa contadores.

Vou bater novamente nessa tecla mostrando outra ilustração (Figura 4.12) pois acho importante que se entenda a ideia de o que é um *loop*. O fluxo normal é de cima pra baixo linha a linha, mas o *loop* faz com que o fluxo volte pra o início do bloco de instruções definido caso a condição seja atendida (verdadeira) e execute tudo de novo.

4.2.1 *do...while*

A primeira estrutura é o *do...while* que você já viu nos exemplos anteriores. Observe que a condição fica apenas no fim da estrutura e essa é a principal característica e o diferencial em relação às outras, uma vez que essa disposição **garante** que o bloco seja executado pelo menos uma vez. Já nas outras, como a condição fica no início, se por acaso ela não for satisfeita de cara o programa nem entra no loop, já pula pra depois da estrutura. Veja a ilustração dessa estrutura na Figura ??.

Sem mais delongas, mão na massa: vamos começar com um exemplo simples, que consiste em pedir 10 números e depois mostrar a soma deles. Já mostrei antes a estrutura do *do...while*, então tenta fazer ai antes de ver a resposta (Figura 4.14). Nessa resposta usei conceitos como o de “+=”, presentes na Seção 3.2.7.

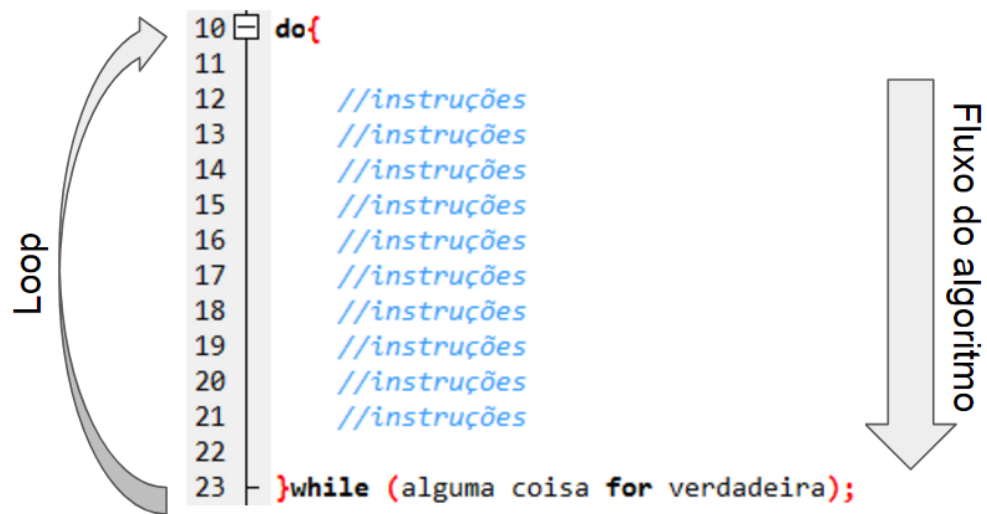


Figura 4.12: Fluxograma de um loop. Fonte: Autor

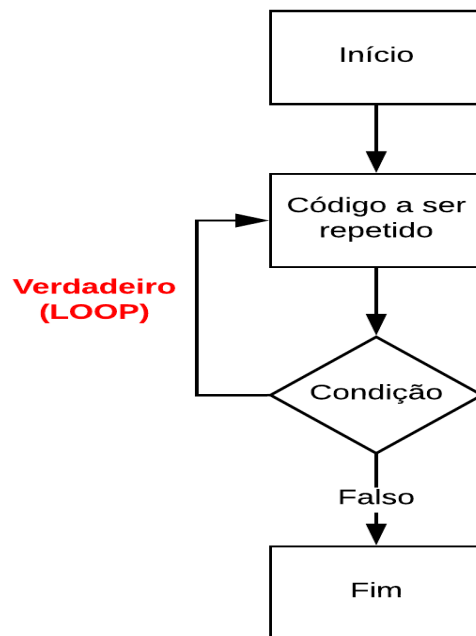


Figura 4.13: Fluxograma do *do...while*. Fonte: Autor

```

1  #include <iostream>
2
3  //Programa que pede e mostra a soma de 10 números
4
5  int main(){
6      using namespace std;
7
8      int contador = 1; //contará os loops
9      int soma = 0; //guardará a soma dos números
10
11     int numero; //guardará o número que foi digitado
12     //observe que não precisamos guardar todos os números, apenas a soma deles
13
14     do{
15         cout << "Digite o " << contador << "º número -> ";
16         cin >> numero;
17
18         soma += numero; //mesma coisa que -> "soma = soma + numero"
19
20         contador++; //adicionando uma unidade ao contador
21
22     }while(contador <= 10);
23
24     cout << "A soma foi = " << soma;
25
26     return 0;
27 }

```

Figura 4.14: Programa que mostra a soma de 10 números informados pelo usuário. Fonte: Autor

Detalhe importante: observe que temos que colocar um ponto e vírgula no fim dessa estrutura! É uma particularidade do *do...while*. Outra coisa: na linha 15 eu coloquei o contador no meio do cout pra ficar uma saída bonitinha: fica tipo “Digite o 1º número -> ” ai depois “Digite o 2º...”. Teste e veja que fica legal.

4.2.2 *while*

Parecido com a estrutura anterior, a diferença é que a condição fica no início do bloco de comandos, o que, ao contrário da anterior, não garante que o algoritmo execute o que tem dentro do loop. Talvez o fluxograma da Figura 4.15 deixe isso mais claro pra você.

Observe que a figura deixa bem clara a principal diferença entre o *while* e o *do...while*: por conta da posição da condição em relação ao bloco, ela deve ser atendida logo de cara pra o bloco de comandos ser executado pelo menos uma vez, **se não for atendida o programa nem entra no bloco**. Como na estrutura anterior a condição fica depois do bloco, **ele sempre vai ser executado pelo menos uma vez**. É muito importante notar essa diferença na hora de codificar.

Agora, finalmente a prática: crie um algoritmo que peça números. Caso o usuário digite 0, ele para de pedir números e mostra a soma de todos os números digitados. Esse é um exemplo que não usa contador e, portanto, exige um olhar diferente sobre o algoritmo. Tente fazer e depois veja a resposta na Figura 4.16.

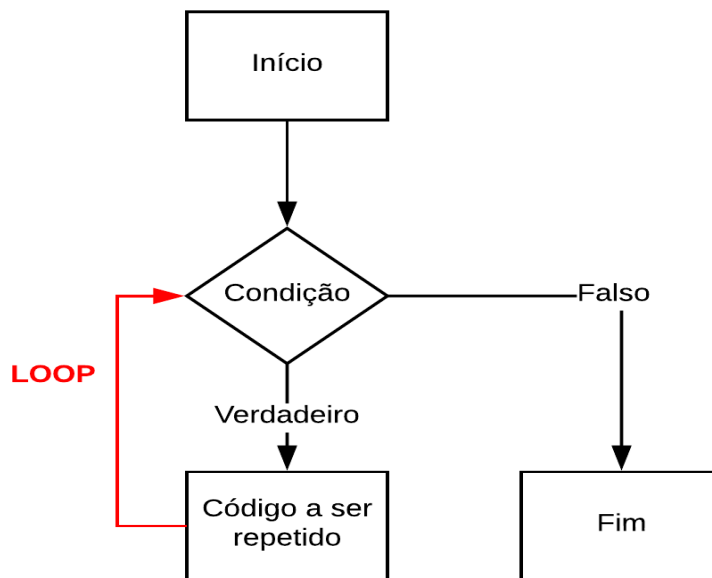


Figura 4.15: Fluxograma do *while*. Fonte: Autor

```

1  #include <iostream>
2
3  //Programa que pede e mostra a soma de números até que o usuário digite 0
4
5  int main(){
6      using namespace std;
7
8      int soma = 0; //guardará a soma dos números
9      int numero; //guardará o número a ser somado
10
11     cout << "Digite um valor (0 para encerrar): ";
12     cin >> numero;
13
14     while(numero!=0){
15         soma += numero; //mesma coisa que -> "soma = soma + numero"
16
17         cout << "Digite outro valor (0 para encerrar): ";
18         cin >> numero;
19     }
20
21     cout << "A soma dos números digitados foi = " << soma;
22
23     return 0;
24 }
25

```

Figura 4.16: Programa que soma de números informados pelo usuário até que o número seja igual a 0. Fonte: Autor

Observe o seguinte: se o usuário digitar 0 logo na primeira leitura, o programa nem entra no *while*, já que a condição já começa não sendo satisfeita (*x* é igual a 0 logo de cara). Tá aí na prática a principal diferença entre o *do...while* e o *while*.

Além disso, observe que, independente da quantidade de números, eu só usei uma variável pra guardá-los, pois não precisarei saber quais números foram digitados, apenas a soma deles. São valores temporários que podem ser substituídos pelos próximos.

4.2.3 *For*

Por último mas não menos importante, o meu favorito: o *for*. O diferencial dele é que ele é visualmente mais organizado que os outros, já que concentra os “mecanismos” da repetição logo no início e no mesmo lugar. Os mecanismos já foram apresentados, são eles: a condição, o incremento (contador) e a inicialização da variável de contagem (esse último vai ficar mais claro na prática). Veja abaixo a estrutura do *for*:

```
for(inicialização; condição; incremento){
    //instruções
}
```

O que antes ficava no meio das instruções (a gente colocava *x++* em qualquer canto dentro do *while*, por exemplo), agora DEVE ficar logo no início entre os parêntesis do *for* e na ordem apresentada. Caso você precise mudar algum desses parâmetros é muito mais fácil de achar e de fazer a manutenção. Obviamente, o incremento pode ser de qualquer valor: se quiser, por exemplo, aumentar de 7 em 7 é só colocar *x+=7*, se quiser diminuir basta usar *x-=7*. Mais fácil que dar tapa em bêbado (não que ele mereça isso).

Exemplificando os mecanismos, poderíamos ter

```
for(int x=0; x<=10; x++){
    cout << x << endl;
}
```

para mostrar os números de 0 a 10. Ou então:

```
for(int x=10; x>=0; x--){
    cout << x << endl;
}
```

para mostrar os números de 10 a 0. Massa né?

Você também pode colocar vários parâmetros juntos. Vamos ver um *for* que vai de 0 a 10 e de 10 a 0 ao mesmo tempo:

```
int x,y;
for(x=0, y=10; x<=10 && y>=0; x++, y--){
    cout << x << " | " << y << endl;
}
```

Enquanto o x vai diminuindo, o y vai crescendo ao mesmo tempo.

Mas observe que nem sempre o *for* é a melhor opção pra tudo. *Loops* que não usam contador (Figura 4.16), ou seja, que não tem um número pré-determinado de *loops*, podem ser melhores com uso das outras estruturas. Você tem livre escolha, mas fuja de gambiarras e tente deixar seu código sempre o mais limpo e inteligente possível.

Partindo pra prática, lá atrás, na Figura 4.5 eu mostrei como determinar se um número é par ou ímpar, e na Figura 4.11 eu mostrei como exibir os números de 1 a 50, agora é só “juntar” os dois conhecimentos em um algoritmo só para poder mostrar os primeiros 50 números pares. Antes de ver a resposta (Figura 4.17), passe uma boa meia hora tentando fazer isso.

```
1  #include <iostream>
2
3  //Programa que mostra os 50 primeiros pares
4
5  int main(){
6      using namespace std;
7
8      int x;
9
10     for(x=0; x<=50; x++){
11         if(x%2 == 0){ //se o resto da divisão de x por 2 for == 0 (se x for par)...
12             cout << x << endl; //mostre x
13         }
14     }
15
16     return 0;
17 }
```

Figura 4.17: Os primeiros 50 números pares. Fonte: Autor

Mas pense direitinho: concorda comigo que se eu somar 2 a qualquer número ímpar o resultado é o próximo ímpar? Sabendo que o incremento do *for* pode ser de qualquer valor (“y++” soma uma unidade, mas você pode escrever “y+=10” e somar 10, por exemplo), como poderíamos fazer o algoritmo anterior sem usar *if* e, conseqüentemente, de forma mais eficiente e inteligente? Pense, tente algumas vezes e o resultado tá na Figura 4.18.

```
1  #include <iostream>
2
3  //Programa que mostra os 50 primeiros pares
4
5  int main(){
6      using namespace std;
7
8      int x;
9
10     for(x=2; x<=50; x+=2){
11         cout << x << endl;
12     }
13
14     return 0;
15 }
```

Figura 4.18: Os primeiros 50 números pares de forma mais eficiente e inteligente. Fonte: Autor

4.3 Funções

DRAFT

Parte II

Caso de Uso: Simulação de Sistemas Usando Autômatos Celulares e C++

Parte III

Paralelizando o Processamento dos Dados com MPI

Lista de Figuras

2.1	Clique no botão verde pra baixar, simples assim. Fonte: Autor	14
2.2	Detalhe apenas para o tipo de instalação, pode escolher a <i>Full</i> (completa) mesmo. Fonte: Autor	15
2.3	Esta é a interface do programa sem um projeto aberto. Inicie um novo da forma mostrada acima. Fonte: Autor	15
2.4	Configurações de novos projetos. Siga esse modelo pra configurar o seu. Fonte: Autor	16
2.5	“Esqueleto” de um programa em C++. Fonte: Autor	16
3.1	<i>Hello World!</i> . Fonte: Autor	17
3.2	<i>Compilando nosso programa</i> . Fonte: Autor	18
3.3	<i>Hello World</i> em execução. Fonte: Autor	18
3.4	Ilustração da memória RAM. Fonte: Autor	21
3.5	Primeira e mais detalhada forma de se mostrar o dobro de 10. Fonte: Autor	22
3.6	Forma mais simples de mostrar o dobro de 10. Fonte: Autor	23
3.7	Programa do dobro em execução. Fonte: Autor	23
3.8	Programa do dobro em execução usando locale. Fonte: Autor	24
3.9	Programa que exemplifica o uso das variáveis Fonte: Autor	25
3.10	Programa das variáveis rodando. Fonte: Autor	26
3.11	Programa para exemplificar o uso dos modificadores de tipo. Fonte: Autor	28
3.12	Programa dos modificadores executando, respectivamente, no Windows e no Ubuntu. Fonte: Autor	28
3.13	Cuidados no uso das variáveis. Fonte: Autor	30
3.14	Lendo e somando 2 números. Fonte: Autor	30
3.15	Programa que diferencia operadores pré e pós fixados. Fonte: Autor	32
3.16	Você quer somar dois números, @?. Fonte: Autor	33
3.17	Você quer somar dois números, @? (versão melhorada). Fonte: Autor	34
3.18	Constantes na prática. Fonte: Autor	35
4.1	Fluxograma para exemplificar os “caminhos” que um programa pode seguir após uma tomada de decisão. Fonte: Autor	37
4.2	Programa que define o maior entre dois números. Fonte: Autor	37
4.3	Operadores comparativos. Fonte: Autor.	38
4.4	Operadores lógicos. Fonte: Autor.	40
4.5	Par ou ímpar?. Fonte: Autor.	40

4.6	Maior de dois números com operador ternário e com <i>if...else</i> . Fonte: Autor	41
4.7	Simulação de um caixa automático para exemplificar o uso do <i>switch</i> . Fonte: Autor.	42
4.8	Porque usar o <i>break</i> . Fonte: Autor	43
4.9	Calculadora com <i>switch</i> . Fonte: Autor	44
4.10	Como usar as chaves. Fonte: Autor	45
4.11	Mostrando os números de 1 até 50. Fonte: Autor	46
4.12	Fluxograma de um loop. Fonte: Autor	47
4.13	Fluxograma do <i>do...while</i> . Fonte: Autor	47
4.14	Programa que mostra a soma de 10 números informados pelo usuário. Fonte: Autor	48
4.15	Fluxograma do <i>while</i> . Fonte: Autor	49
4.16	Programa que soma de números informados pelo usuário até que o número seja igual a 0. Fonte: Autor	49
4.17	Os primeiros 50 números pares. Fonte: Autor	51
4.18	Os primeiros 50 números pares de forma mais eficiente e inteligente. Fonte: Autor	51

DRAFT

Lista de Tabelas

3.1	Tipos de dado. Fonte: https://www.tutorialspoint.com/cplusplus/cpp_data_types.htm . Acessado em: 12/03/2017.	24
3.2	Modificadores de Tipo	27
4.1	Operadores lógicos	39

DRAFT