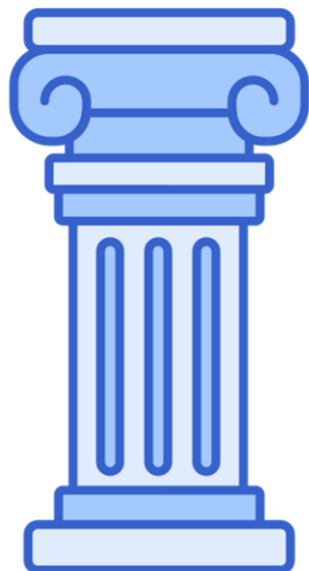


# O Pilar do C



Hadel Rachid Daher Junior  
CUROS HOTONLINE [hadelrachid@gmail.com](mailto:hadelrachid@gmail.com)

## Sumário

CAPÍTULO 1 .....	1
1.1. Escrevendo um Programa C++ Simples .....	1
1.1.1. Compilando e executando o nosso programa .....	3
Uma breve explicação sobre o Cygwin e o MSYS2 .....	5
Um compilador gratuito de C e C++ para Microsoft Windows .....	6
Construção autônoma do WinLibs de GCC e MinGW-w64 para Windows.....	6
1.1.1.1. Configurar o Visual Studio Code para programação em C/C++ .....	20
1.1.1.2. A coleção de compiladores GNU e o projeto LLVM .....	1
Sistemas Linux .....	2
Compiladores de linha de comando para PC .....	3
Ambientes de desenvolvimento integrados – IDE (Windows).....	4
Capítulo 2.....	1
2.1. Um exemplo simples de C .....	1
O exemplo explicado .....	1
Passo 1: Sinopse Completa.....	2
#include <stdio.h> .....	2
Como fazer comentários para documentação: /* um programa simples */ ...	2
Início do corpo da função, que seria o abre chaves: {.....	2
Declarando o nome de uma variável e seu tipo: int num; .....	2
Fazendo uma atribuição de um valor inteiro a uma variável, no caso, “num” que recebe o valor inteiro 1. ....	3
Uma instrução de chamada de função: printf("Eu sou um simples");.....	3
Outra instrução de chamada de função: printf("computador.\n"); .....	3
printf("Meu número favorito é %d porque é o primeiro.\n", num); .....	4
Uma declaração de retorno para a função do tipo inteiro: return 0; .....	4
Fim da função main(): }.....	4
Passo 2: Detalhes do Programa .....	4
#include Diretivas e Arquivos de Cabeçalho .....	4
A função principal main() .....	5
Comentários .....	6
Chaves, corpos e blocos .....	8
Declarações.....	8
Tipos de dados.....	10

Escolha do nome .....	10
Quatro boas razões para declarar variáveis .....	11
Atribuição .....	12
A função printf() .....	13
Declaração de retorno .....	14
A estrutura de um programa simples.....	15
Dicas para tornar seus programas legíveis.....	15
Dando mais um passo no uso de C.....	17
Documentação.....	18
Múltiplas Declarações .....	19
Multiplicação .....	19
Imprimindo vários valores .....	19
Enquanto você faz isso – múltiplas funções.....	20
RESUMINDO: .....	22
1. Verificação de Tipos.....	22
2. Declarações Antecipadas.....	23
3. Modularidade e Organização .....	23
4. Documentação.....	23
5. Facilita a Detecção de Erros.....	23
EXEMPLO DE MODULARIAÇÃO .....	24
Passo 1: Arquivo de Cabeçalho (soma.h).....	24
Passo 2: Arquivo de Implementação da Função (soma.c).....	24
Passo 3: Arquivo Principal (main.c).....	25
Compilação .....	25
Explicação: .....	25
Vantagens: .....	26
EXEMPLO DE UM PROJETO USANDO UMA ESTRUTURA DE DIRETÓRIOS.....	26
Estrutura de Diretórios .....	26
Passo a Passo para Compilar e Linkar Usando GCC.....	27
Explicação .....	34
Mais sobre as flags do GCC.....	34
Sobre a flag -fPIC .....	35
Como a flag -fPIC Funciona no Linux.....	35
Quando Usar -fPIC No Linux.....	36
Uso do -fPIC No Windows .....	36

Conclusão.....	36
Exemplo no Windows (sem -fPIC).....	37
Finalizando.....	38
C Primer Plus:.....	3
C Como Programar (Nova Edição Atualizada): .....	3



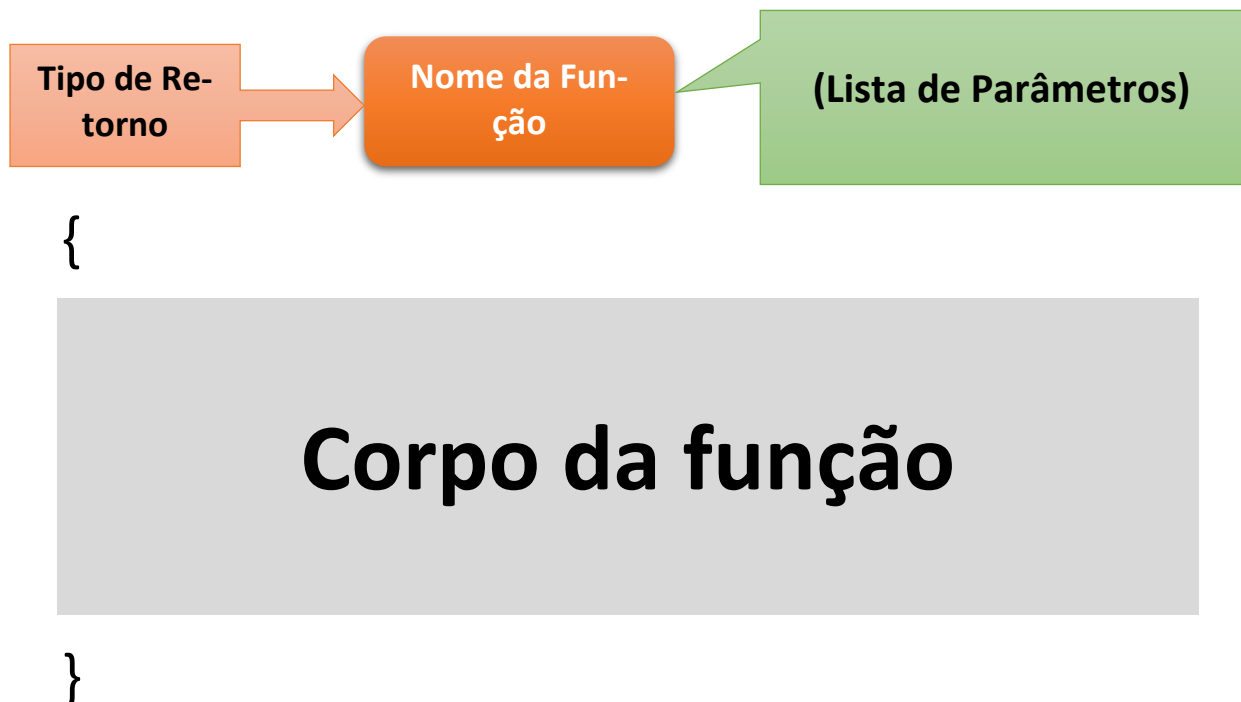
# CAPÍTULO 1

## 1.1. Escrevendo um Programa C++ Simples

Todo programa C++ contém uma ou mais funções, uma das quais deve ser chamada de main. O sistema operacional executa um programa C++ chamando main. Aqui está uma versão simples de main que não faz nada além de retornar um valor para o sistema operacional:

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

Uma definição de função tem 4 (quatro) elementos: **um tipo de retorno**, **um nome de função**, **uma lista de parâmetros** (possivelmente vazia) **entre parênteses** e **um corpo de função**. Embora **main** seja especial em alguns aspectos, definimos **main** da mesma forma que definimos qualquer outra função.



## Definição de função:

- Um tipo de retorno;
- Um nome de função;
- Uma lista de parâmetros entre parênteses;
- Um corpo de função.

Neste exemplo, **main** tem uma lista vazia de parâmetros (mostrado pelo () sem nada dentro).

A função principal deve ter um tipo de retorno **int**, que é um tipo que representa **números inteiros (Z)**. O tipo **int** é um tipo de dado primitivo, ou um bloco de construção básica, o que significa que é um dos tipos definidos pela linguagem.

A parte final de uma definição de função, o corpo da função, é um bloco de declarações começando com uma chave aberta e terminando com uma chave fechada:

```
{ //Chave aberta
    // ...
    // Aqui é o corpo da função
    // onde se faz o bloco de declarações
    // ...
    return 0;
} // Chave fechada
```

A única instrução neste bloco é um retorno, que é uma instrução que finaliza uma função. Como é o caso aqui, um retorno também pode enviar um valor de volta para o chamador da função. Quando uma instrução de retorno inclui um valor, o valor retornado deve ter um tipo compatível com o tipo de retorno da função. Nesse caso, o tipo de retorno de main é **int** e o valor de retorno é **0**, que é um **int**.



*Observe o ponto e vírgula no final da instrução **return**. O ponto-e-vírgula marca o final da maioria das instruções em C++. Eles são fáceis de ignorar, mas, quando esquecidos, podem levar a misteriosas mensagens de erro do compilador.*

Na maioria dos sistemas, o valor retornado de main é um indicador de status. Um valor de retorno de 0 indica sucesso. Um retorno diferente de zero tem um significado definido pelo sistema.

Normalmente, um retorno diferente de zero indica que tipo de erro ocorreu.



### **Conceito-chave: Tipos**

*Tipos são um dos conceitos mais fundamentais em programação e um conceito ao qual voltaremos várias vezes nesta cartilha. Um tipo define o conteúdo de um elemento de dados e as operações possíveis nesses dados.*

*Os dados que nossos programas manipulam são armazenados em variáveis e cada variável tem um tipo. Quando o tipo de uma variável chamada  $v$  é  $T$ , geralmente dizemos que " $v$  tem tipo  $T$ " ou, de forma intercambiável, que " $v$  é um  $T$ ".*

## **1.1.1. Compilando e executando o nosso programa**

Tendo escrito o programa, precisamos compilá-lo.

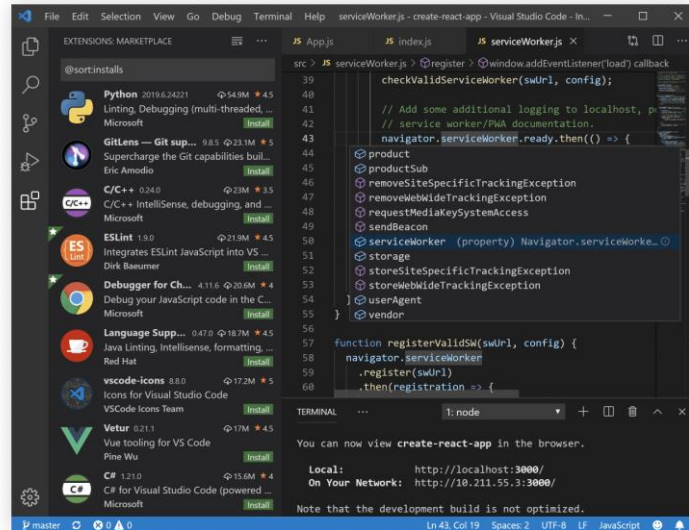
Como você compila um programa depende do seu sistema operacional e compilador. Para obter detalhes sobre como seu compilador específico funciona, verifique o manual de referência ou pergunte a um colega experiente.

Muitos compiladores baseados em PC são executados a partir de um **ambiente de desenvolvimento integrado (IDE - Integrated Development Environment)** que agrupa o compilador com ferramentas de compilação e análise. Esses ambientes podem ser um grande trunfo no desenvolvimento de grandes programas, mas requerem um pouco de tempo para aprender como usá-los de forma eficaz. Aprender a usar tais ambientes está muito além do escopo desta explicação.

A maioria dos compiladores, incluindo aqueles que vêm com um **IDE**, fornece uma interface de linha de comando. A menos que você já conheça o **IDE**, pode achar mais fácil começar com a interface de linha de comando. O **Microsoft Visual Studio** (<https://code.visualstudio.com>), que pode ser uma IDE completa, o qual também possui plugins para outras linguagens, como **C# (C-sharp)**, **Python**, **Java**, entre outras, proporciona essa experiência. O uso de interface de linha de comando permitirá que você se concentre primeiro em aprender **C++**. Além disso, depois de entender a linguagem é provável que o **IDE** seja mais fácil de aprender.

Apesar de não ser uma **IDE**, esse editor, que será utilizado nas explicações, é uma ferramenta útil para projetos de pequenas aplicações, no meu caso, estou utilizando o **editor de código-fonte** desenvolvido pela **Microsoft** para sistemas **Windows**, **Linux** e **MacOS**, que é o **Visual Studio Code**.



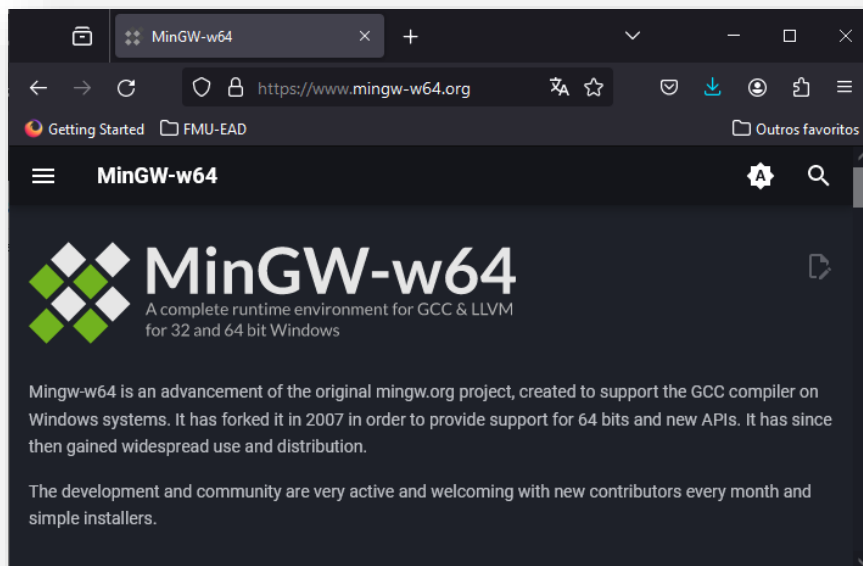


*Figura 1. Visual Studio Code.*

**Baixe o Editor no site:**

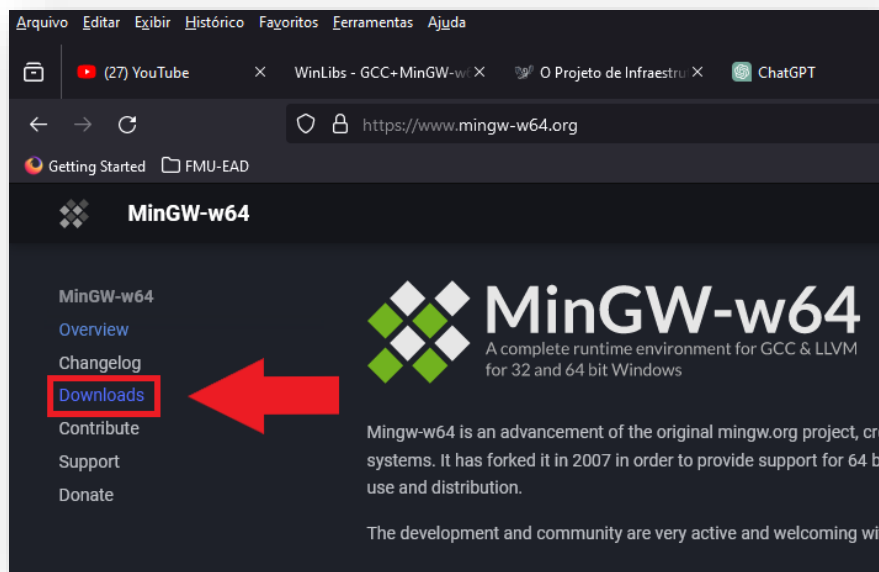
<https://code.visualstudio.com/download>

Também estou utilizando o sistema operacional **windows 10**, nesta explicação, no meu caso, **64-bits**, e o compilador **g++**, para **64-bits**, baixado no site <https://www.mingw-w64.org/> . Se seu sistema for de **32-bits**, você deve baixar o compilador **g++** para **32-bits**.



*Figura 2. Site para baixar o MinGW.*

Então vá em Downloads, e baixe a versão para o seu sistema. Veja a **Figura 3**.



*Figura 3. Link para Download.*

Na página de Downloads do site, temos o **Cygwin**, o **MSYS2** e o **LLVM-MinGW**.

### *Uma breve explicação sobre o Cygwin e o MSYS2*

A principal característica do **Cygwin** é sua capacidade de emular funcionalidades típicas de sistemas operacionais **Unix** em um ambiente **Windows**. Isso é alcançado fornecendo uma biblioteca de runtime que emula as chamadas de sistema e a funcionalidade da API encontrada em sistemas Unix.

Além disso, o **Cygwin** inclui uma coleção de ferramentas de desenvolvimento, como compiladores **GCC**, interpretadores de linguagens como **Python** e **Perl**, e muitas outras ferramentas de linha de comando comuns em sistemas **Unix**. Ao utilizar o **Cygwin**, os desenvolvedores podem escrever e compilar código para **Unix-like** em um ambiente familiar, mesmo quando estão trabalhando em um sistema **Windows**. Isso é especialmente útil para desenvolvedores que estão acostumados a trabalhar com ferramentas e ambientes Unix e desejam manter a mesma experiência de desenvolvimento ao trabalhar em projetos **Windows**.

O **Cygwin** é uma ferramenta valiosa para desenvolvedores que desejam desenvolver e executar aplicativos Unix-like em sistemas Windows, fornecendo uma camada de compatibilidade **POSIX** e uma coleção abrangente de ferramentas de desenvolvimento.

Já o **MSYS2** é uma plataforma de desenvolvimento para **Windows** que fornece um ambiente **Unix-like** para desenvolvedores. Ele é uma continuação do projeto **MSYS (Minimal SYStem)**, mas com uma série de melhorias e atualizações significativas.

O objetivo principal do **MSYS2** é fornecer um ambiente de linha de comando similar ao encontrado em sistemas **Unix**, mas adaptado para o **Windows**. Isso é alcançado fornecendo um conjunto de ferramentas de desenvolvimento, incluindo **shell (bash)**, utilitários de linha de comando (como **ls**, **grep**, etc.), e uma variedade de bibliotecas e ferramentas de compilação, como **GCC**, **make**, **autotools**, entre outros.

Além disso, o **MSYS2** inclui um gerenciador de pacotes chamado **pacman**, que permite aos usuários instalar, atualizar e gerenciar facilmente pacotes de software. Isso torna mais fácil para os desenvolvedores manter seu ambiente de desenvolvimento atualizado com as versões mais recentes das ferramentas e bibliotecas. Uma característica importante do **MSYS2** é a capacidade de instalar e usar pacotes binários do **Arch Linux**, o que significa que os usuários têm acesso a uma ampla variedade de software pré-compilado e podem facilmente instalar novos pacotes conforme necessário.

O **MSYS2** também é frequentemente usado em conjunto com o **MinGW-w64**, que é um conjunto de ferramentas de compilação para Windows baseado no **GCC**. Isso permite que os desenvolvedores construam e executem aplicativos nativos do Windows diretamente no **MSYS2**.

Em resumo, **MSYS2** é uma plataforma de desenvolvimento poderosa para Windows que fornece um ambiente **Unix-like** completo, incluindo um shell, ferramentas de linha de comando e um gerenciador de pacotes, facilitando o desenvolvimento de software em sistemas Windows com uma experiência familiar para desenvolvedores familiarizados com Unix.

## *Um compilador gratuito de C e C++ para Microsoft Windows*

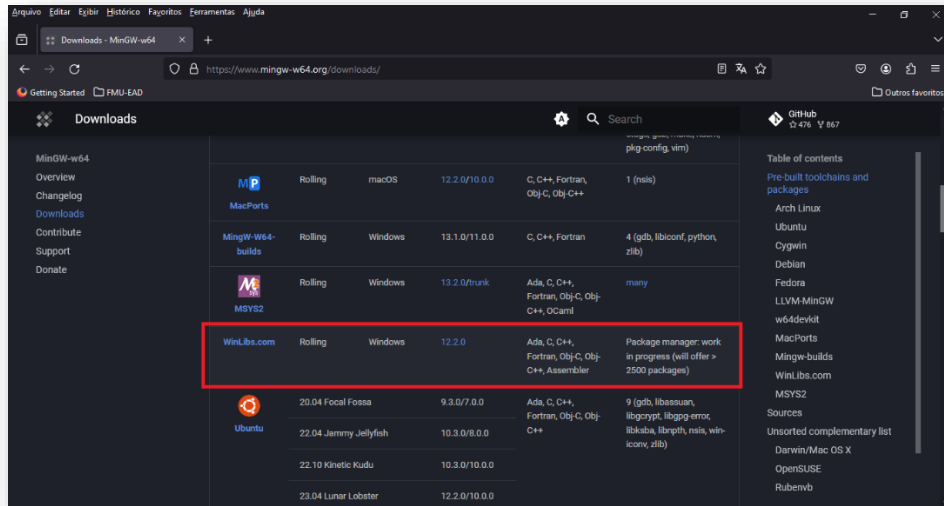
### Construção autônoma do WinLibs de GCC e MinGW-w64 para Windows

**GCC (GNU Compiler Collection)** é um compilador livre e de código aberto para C e C++ (e outras linguagens como Objective-C, Fortran, D).

**MinGW-w64** é uma biblioteca C gratuita e de código aberto para segmentar plataformas Windows de 32 bits e 64 bits.

A combinação destes resulta em um compilador C/C++ gratuito para Windows. Mesmo que o GCC e o MinGW-w64 possam ser usados em outras plataformas (por exemplo, Linux) para gerar executáveis do Windows, o projeto WinLibs só se concentra na construção de versões que são executadas nativamente no Windows.

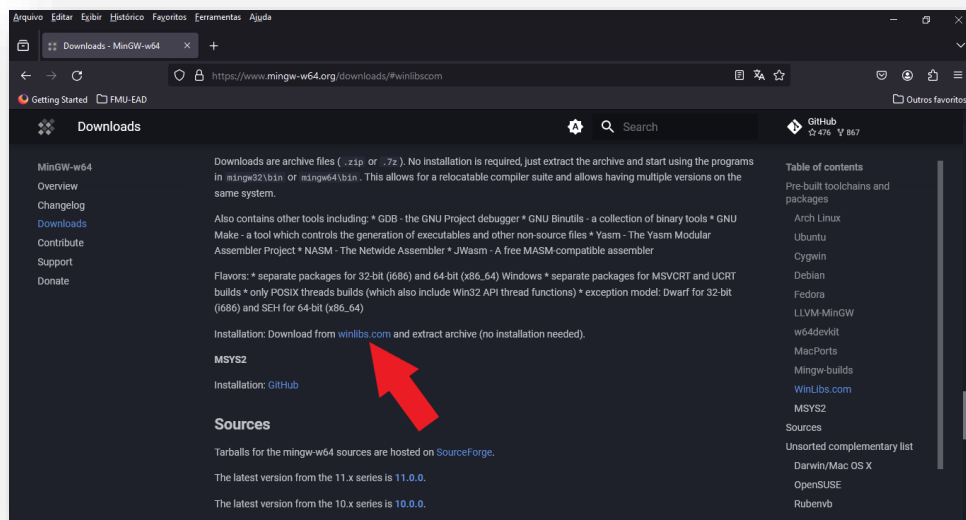
Na minha opinião, por enquanto, como estou utilizando o **Windows 10**, de **64 bits**, mas podendo ser de **32 bits**, o que eu considero o mais prático e o menos burocrático de instalar é o **WinLibs**, visto que ele já possui a terceira opção que o **LLVM-MinGW**. Então role até encontrar a opção [WinLibs.com](https://winlibs.com).



*Figura 4. Role para encontrar a opção WinLibs.*

Acessando a outra página, vá no link [WinLibs.com](https://winlibs.com). Ou acesse o link diretamente daqui!

Link: <https://winlibs.com>



*Figura 5. Link WinLibs.com*

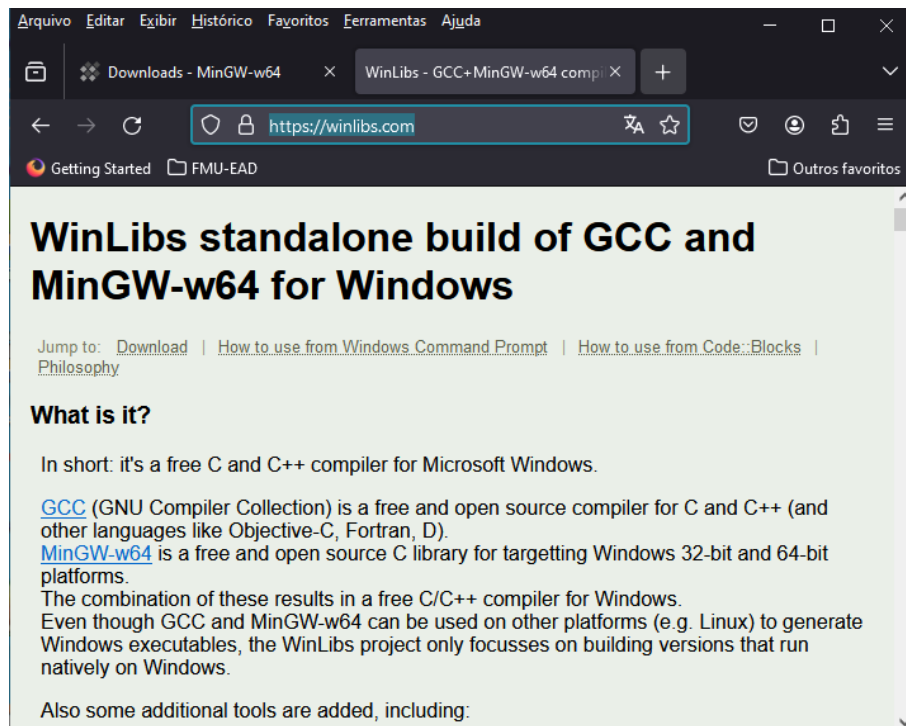


Figura 6. Site do WinLibs.com

Dentro do site do WinLibs.com, role até **Release versions**, no UCRT runtime. Veja a imagem a seguir!

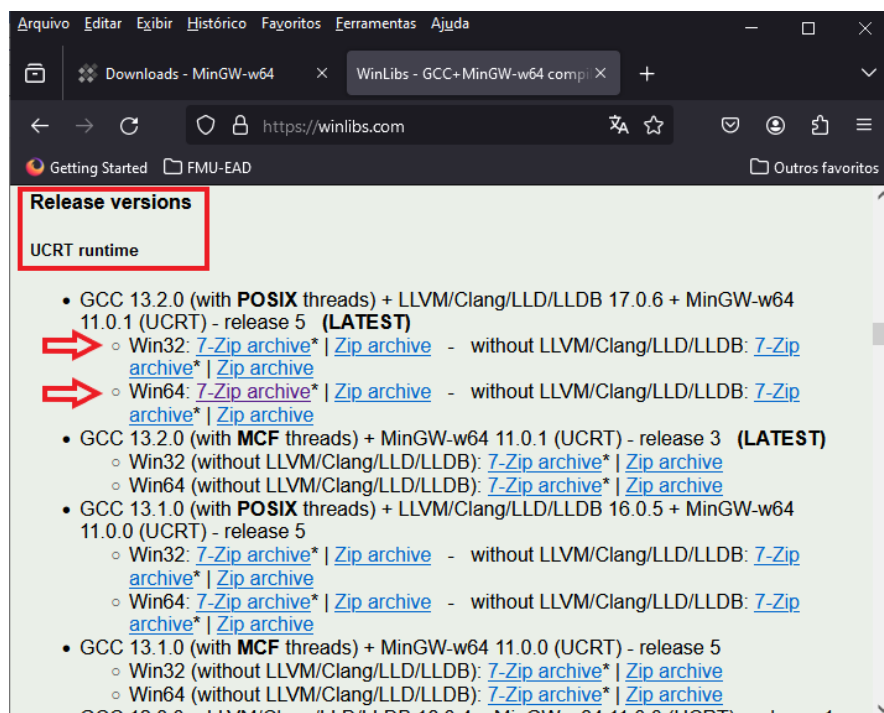
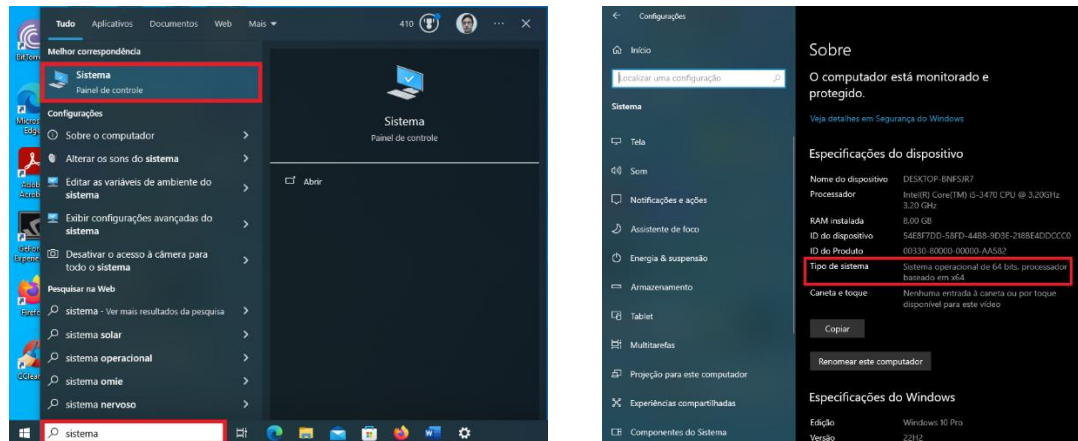


Figura 7. Escolher entre Win32 e Win64, de acordo com o seu sistema.

O meu sistema, na edição desta explicação, estou utilizando o **Windows 10 de 64 bits**. Portanto, vou baixar o **Win64**. Dê preferência ao primeiro link! Porque o segundo é sem o conjunto de ferramentas LLVM/Clang/LLD/LLDB.

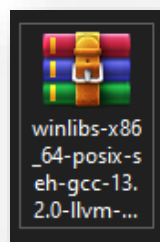
Se no seu caso, o seu sistema for de 32 bits, então baixe o arquivo compactado, com extensão Zip, de 32 bits (Win32). Lembre-se que você deve ter em sua máquina um descompactador de arquivos instalado, podendo ser o [winzip](#) ou o [winrar](#).



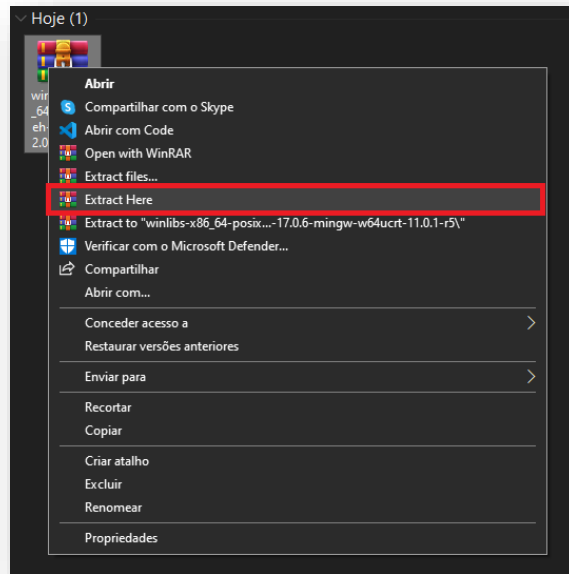
*Figura 8. Sistema e configuração de 32 e 64 bits.*

Se tiver dúvidas quanto ao sistema, vá no seu Windows 10 ou 11, e no local de pesquisar, digite “sistema”. Vai aparecer a opção sistema. Clique sobre ele, e lá aparecerá as informações sobre seu sistema operacional, incluindo as configurações do processador de 32 bits ou 64 bits.

Feito o download do arquivo compactado do WinLibs de 64 bits, com o botão direito do mouse, vá na opção “**Extract Here**” (Extrair Aqui)! Veja as imagens a seguir!



*Figura 9. Pasta de Arquivos do WinLibs compactada.*



*Figura 10.*  
*Com o botão direito do mouse descompacte a pasta mingw64.*

Terminada a descompactação da pasta mingw64, copie, ou recorte, com o botão direito do mouse, ou **CTRL + C** (copiar), ou **CTRL + X** (recortar), esta pasta e depois vá no seu **explorador de arquivos**, veja a **Figura a seguir**.

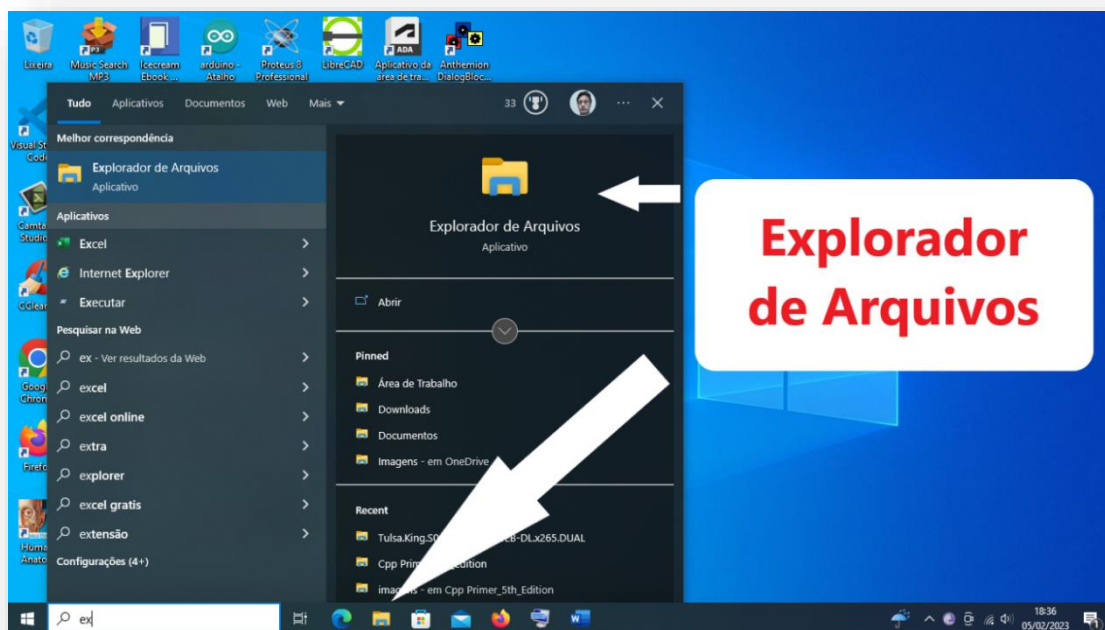


Figura 11. Explorador de Arquivos.

Abrindo o **Explorador de Arquivos**, acesse a opção “**Este Computador**”, ou, em inglês, “**This Computer**”. Veja a Figura 12. E vá na unidade C:

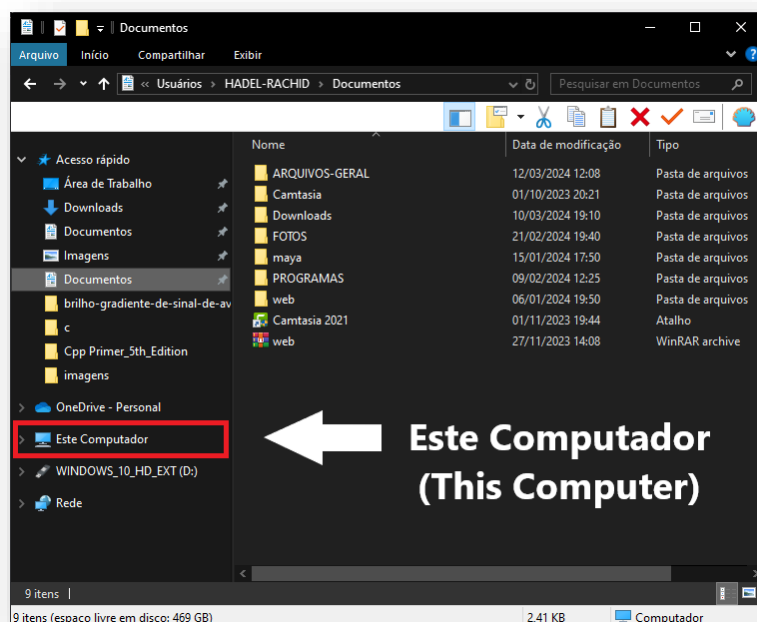
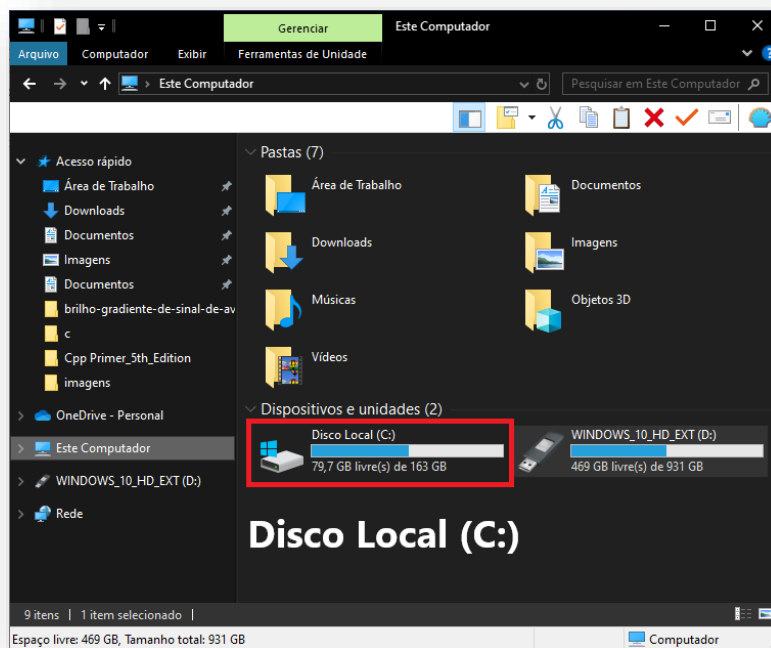


Figura 12. Este Computador.

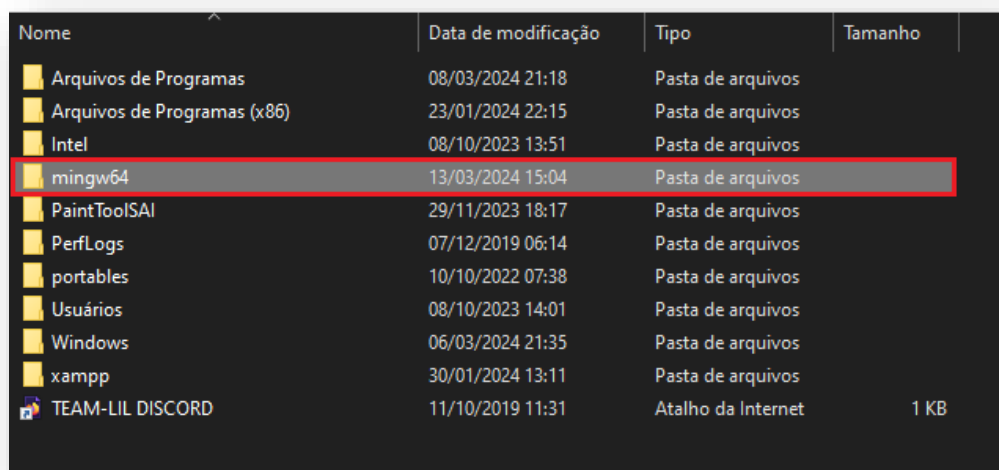


Na próxima imagem, vá na unidade de Disco Local (C:).



*Figura 13. Disco Local (C:).*

Acessando o local da unidade C, cole a pasta mingw64. Veja a imagem abaixo.



*Figura 14.  
Pasta mingw64 transferida para a unidade C.*

Agora acesse a pasta mingw64, e vá na pasta bin. Veja a figura a seguir.

Nome	Data de modificação	Tipo	Tamanho
bin	13/03/2024 15:04	Pasta de arquivos	
include	13/03/2024 15:04	Pasta de arquivos	
lib	13/03/2024 15:04	Pasta de arquivos	
libexec	13/03/2024 15:04	Pasta de arquivos	
share	13/03/2024 15:04	Pasta de arquivos	
x86_64-w64-mingw32	13/03/2024 15:04	Pasta de arquivos	
version_info	03/02/2024 07:01	Documento de Te...	1 KB

*Figura 15. Pasta bin dentro da pasta mingw64.*

O diretório **bin** é o local onde se encontram os compiladores. Então entrando nessa pasta, dando 2 (dois) cliques ou pressionando **Enter**, podemos visualizar, percorrendo pelo interior deste local, os executáveis do **gcc.exe** e do **g++.exe**. Veja a **Figura 16**.

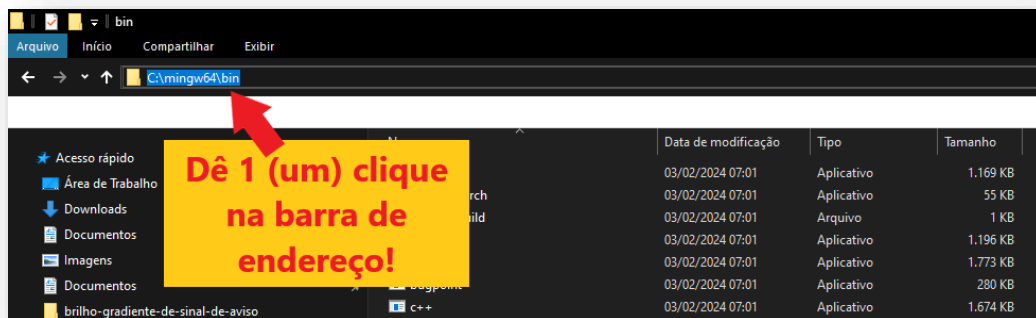
Nome	Data de modificação	Tipo	Tamanho
doxygen	01/02/2024 17:37	Aplicativo	22.483 KB
dsymutil	03/02/2024 07:01	Aplicativo	322 KB
elfedit	03/02/2024 07:01	Aplicativo	75 KB
FileCheck	03/02/2024 07:01	Aplicativo	79 KB
find-all-symbols	03/02/2024 07:01	Aplicativo	150 KB
g++	03/02/2024 07:01	Aplicativo	1.674 KB
gcc	03/02/2024 07:01	Aplicativo	1.671 KB
gcc-ar	03/02/2024 07:01	Aplicativo	72 KB
gcc-nm	03/02/2024 07:01	Aplicativo	72 KB
gcc-ranlib	03/02/2024 07:01	Aplicativo	72 KB
gcov	03/02/2024 07:01	Aplicativo	1.011 KB
gcov-dump	03/02/2024 07:01	Aplicativo	924 KB
gcov-tool	03/02/2024 07:01	Aplicativo	930 KB
gdb	03/02/2024 07:01	Aplicativo	12.912 KB

*Figura 16. Executáveis do gcc.exe e do g++.exe.*

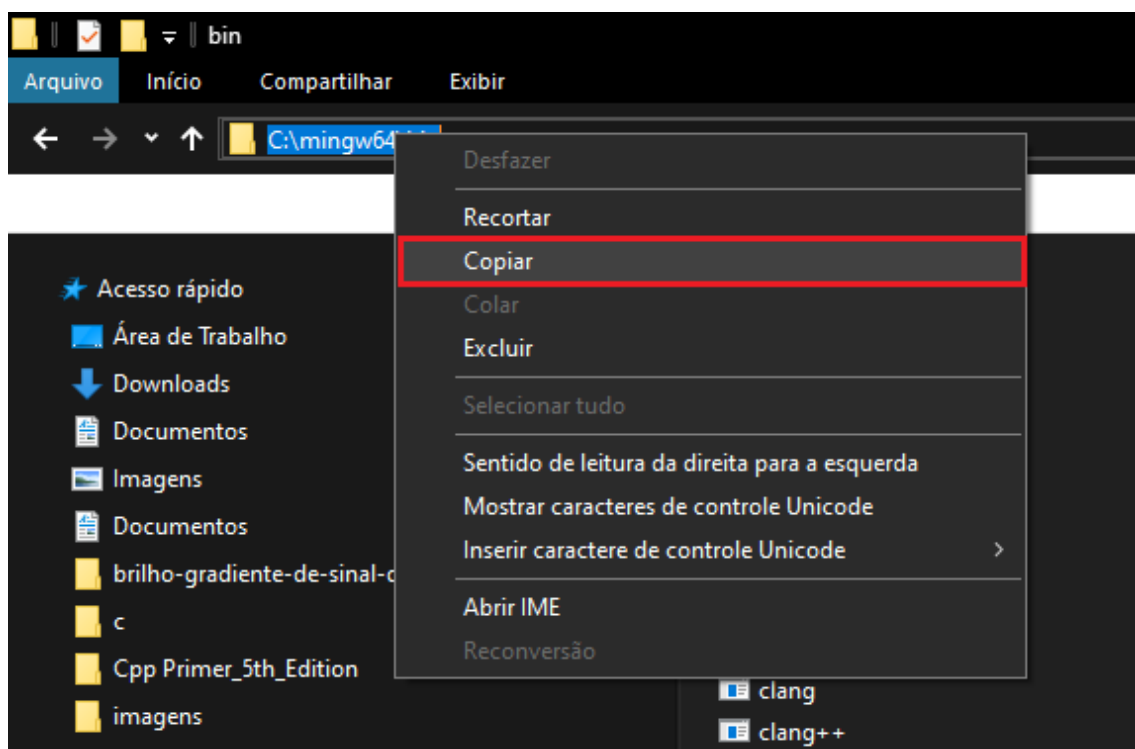
O executável **gcc.exe** será utilizado para compilar arquivos na **linguagem C**. Enquanto o **g++.exe**, para compilar arquivos na linguagem C++.

Para que o **Windows 10** reconheça o caminho destes executáveis, devemos configurar o **Path** no **Editor de Variáveis de Ambiente do Sistema**. Mas antes disso, vamos copiar o local da pasta **bin** do **mingw64**. Então vá na barra de endereço, ainda dentro da pasta

**bin**, dê apenas 1 (um) clique, e com o botão direito do mouse, vá em **copiar**. Na **Figura 17**, o endereço é **C:\mingw64\bin**. Veja as imagens na **Figura 18** e na **Figura 19**.



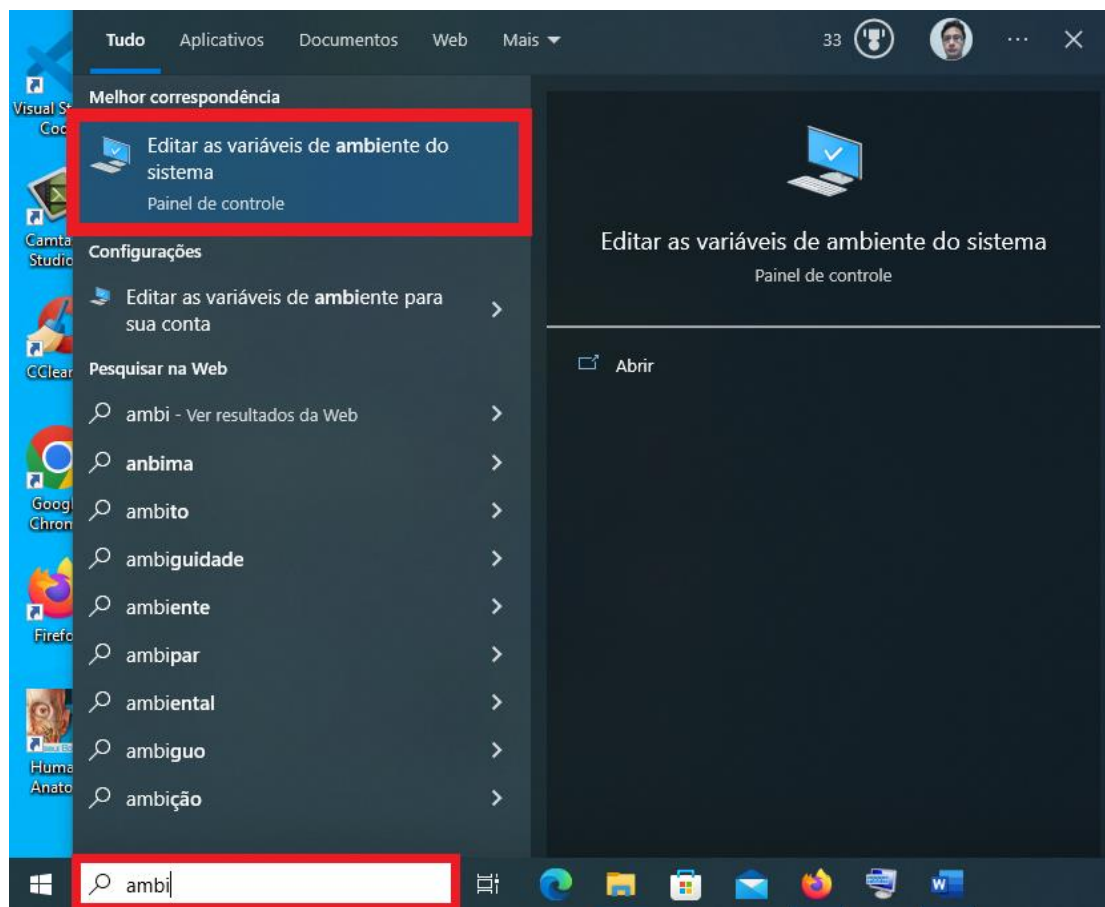
*Figura 17. Barra de Endereço.*



*Figura 18. Copiar o endereço da pasta bin de mingw64.*

Em seguida, vá na barra de pesquisa do **Windows 10** e digite “**Ambiente**”. Veja a **Figura 19**. Aparecerá a opção “**Editar as variáveis de ambiente do sistema**”, então escolha esta alternativa.

Ou você pode ir em **Painel de Controle**, depois em **Sistema** e, por último, **Configurações Avançadas do Sistema**. Neste caso, aparece a janela Propriedades do Sistema.



*Figura 19. Digitar Ambiente. Veja a opção “Editar as variáveis de ambiente do sistema”, no Painel de controle.*

Na **Figura 20**, aparecerá a janela **Propriedades do Sistema**, depois vá no botão **Variáveis de Ambiente**. E na opção **Variáveis do sistema** desta outra Janela, procurar pelo **Path**, onde será configurado o local da pasta **bin** do **mingw64**, para que o sistema operacional reconheça o caminho dos compiladores **gcc.exe** e **g++.exe**. Veja as imagens da **Figura 21** e da **Figura 22**.

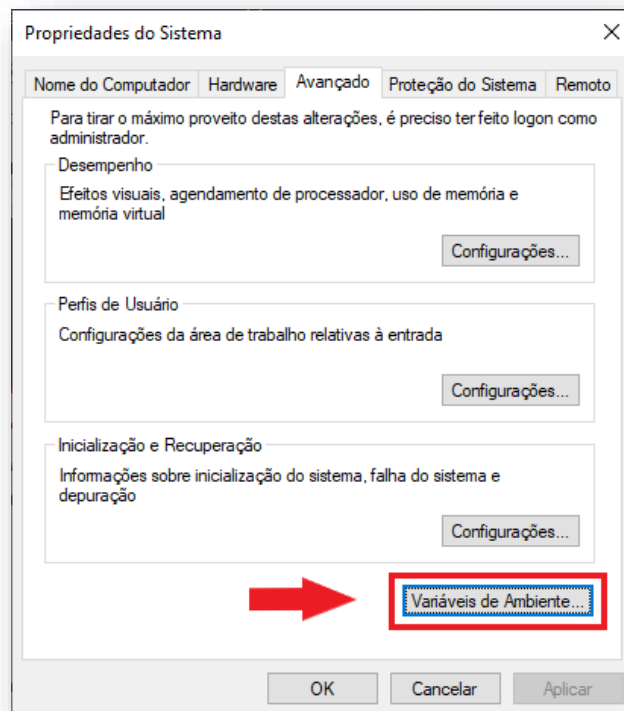


Figura 20. Janela Propriedades do Sistema. Pressione o botão Variáveis de Ambiente.

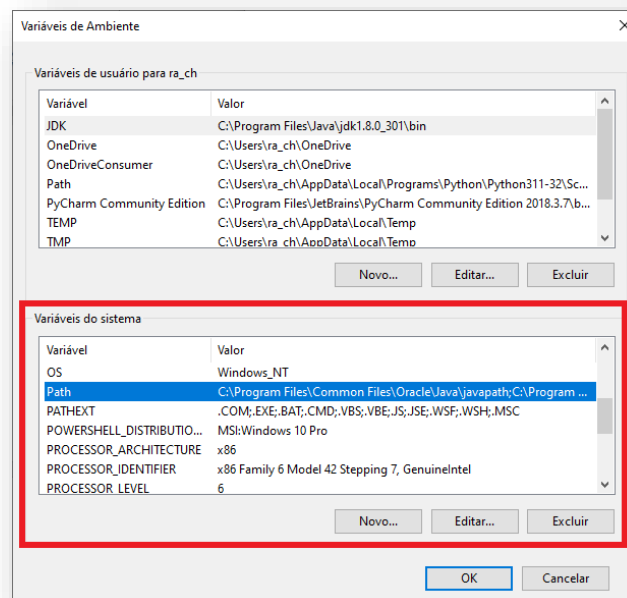
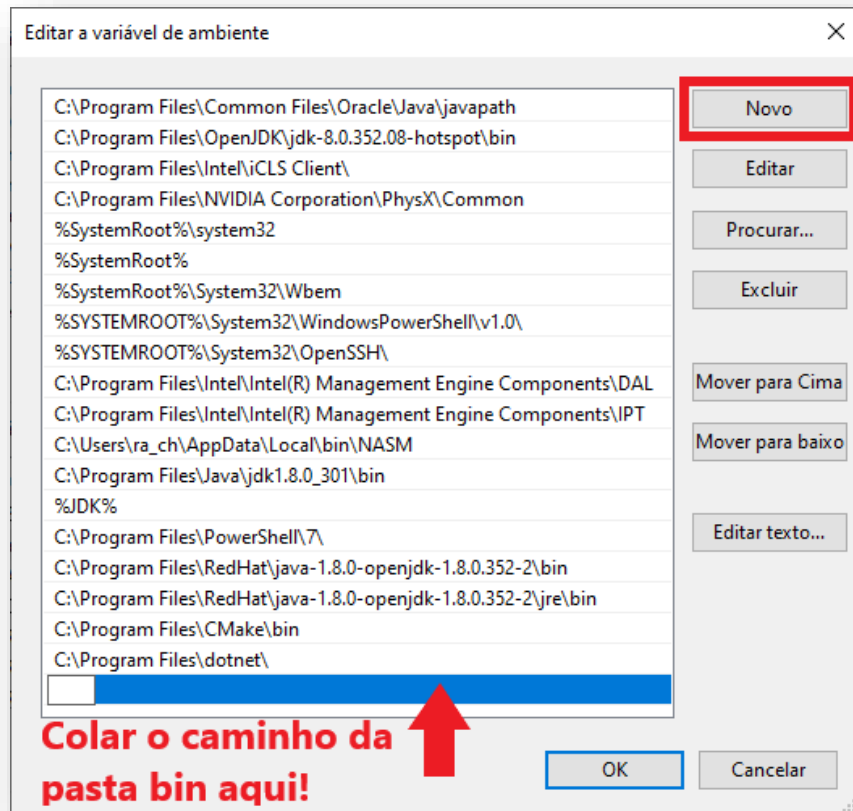
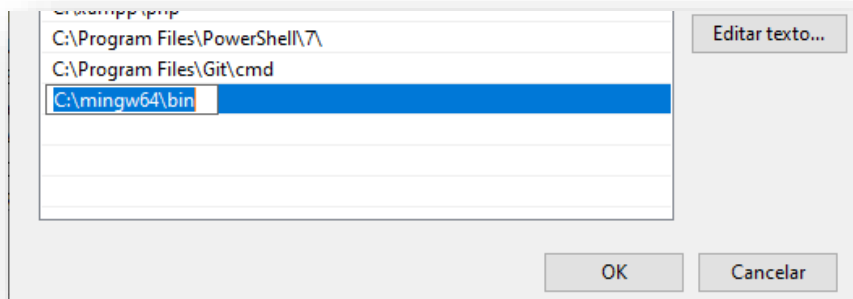


Figura 21. Procurar pelo Path em “Variáveis do sistema”.

Nessa janela, “**Variáveis de Ambiente**”, na opção “**Variáveis do sistema**”, selecione a variável **Path**, depois dê 2 (dois) cliques ou pressione o botão **Editar**. Depois aparecerá uma janela “**Editar a variável de ambiente**”. Em seguida, pressione o botão **Novo**. Aparecerá uma opção para entrada de valor, e com o botão direito do mouse, vá em **colar**, ou pressione no teclado **Ctrl + V**. Veja a **Figura 23**. No meu caso, o caminho ou Path foi: **C:\mingw64\bin**.



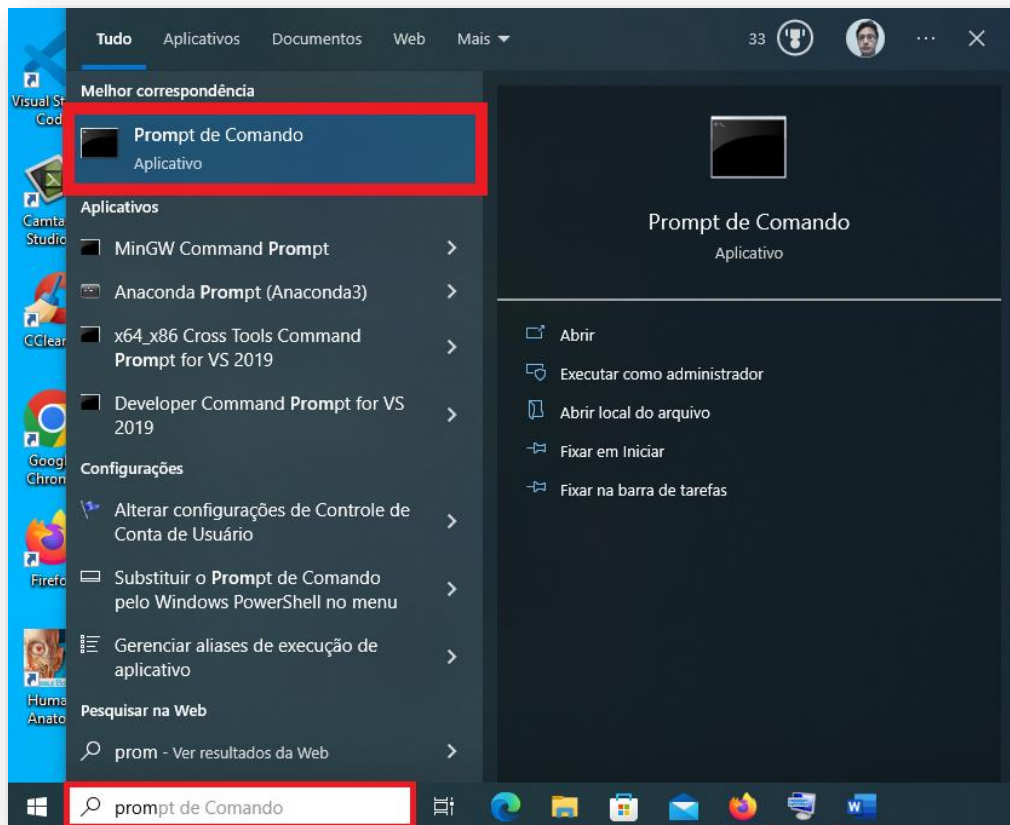
*Figura 22. Colar o caminho na nova entrada de valor.*



*Figura 23. O caminho foi C:\mingw64\bin.*

Para finalizar, pressione **OK**. Dê também **OK** para a janela **Variáveis de ambiente**. E **OK** para a janela **Propriedades do Sistema**.

Agora vamos verificar se o **Windows 10** reconhece o caminho onde o **g++.exe** está localizado para realizar suas tarefas de compilação. Então vá na barra de pesquisa do **Windows 10**, e digite a palavra **Prompt do Comando** ou **Terminal** ou **CMD**, neste caso, tanto faz, porque o **Windows 10** reconhecerá que a escolha é algum terminal de comando. E vá na opção **Prompt do Comando**. Veja a **Figura 24**.



*Figura 24.*

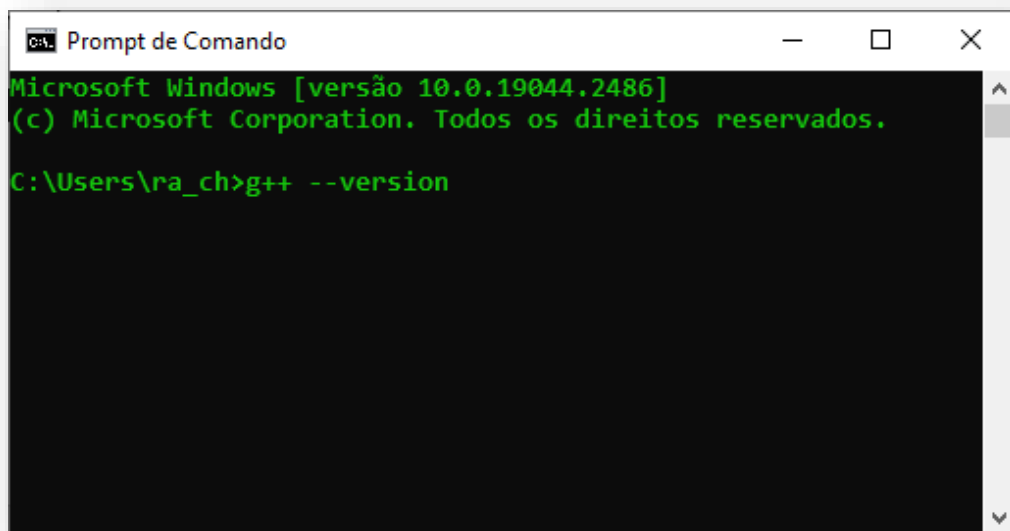
*Digitar Prompt do Comando ou CMD ou Terminal na Barra de Pesquisa do Windows 10.*

Aparecerá a janela do **Prompt do Comando**, a seguir, na **Figura 25**. Então digite o comando abaixo e dê 1 (um) **Enter**, no teclado:

**g++ --version**

ou

**gcc --version**

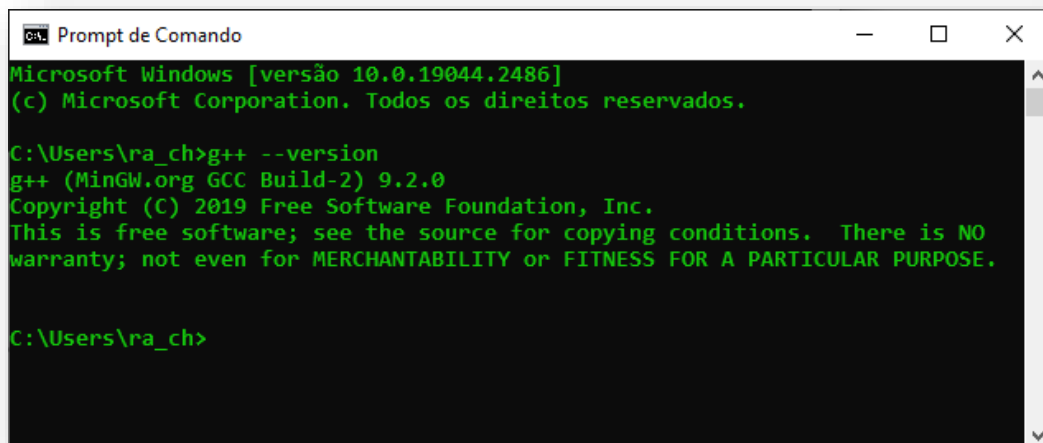


```
CA: Prompt de Comando
Microsoft Windows [versão 10.0.19044.2486]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\ra_ch>g++ --version
```

*Figura 25. Digite o comando: g++ --version.*

Após digitar o comando na **Figura 25**, dê 1 (um) **Enter**. Veja a **Figura 26**.



```
CA: Prompt de Comando
Microsoft Windows [versão 10.0.19044.2486]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\ra_ch>g++ --version
g++ (MinGW.org GCC Build-2) 9.2.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

C:\Users\ra_ch>
```

*Figura 26. Informações da versão do g++.*

Informações da versão do g++
g++ (MinGW-W64 x86_64-ucrt-posix-seh, built by Brecht Sanders, r5) 13.2.0 Copyright (C) 2023 Free Software Foundation, Inc. This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.



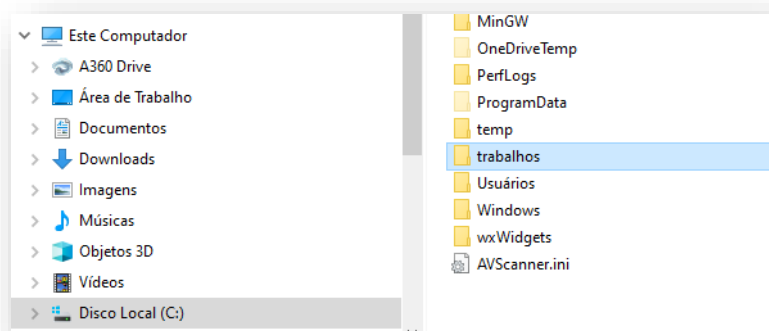
Se aparecer a versão do **g++** como resposta, então tudo deu certo!



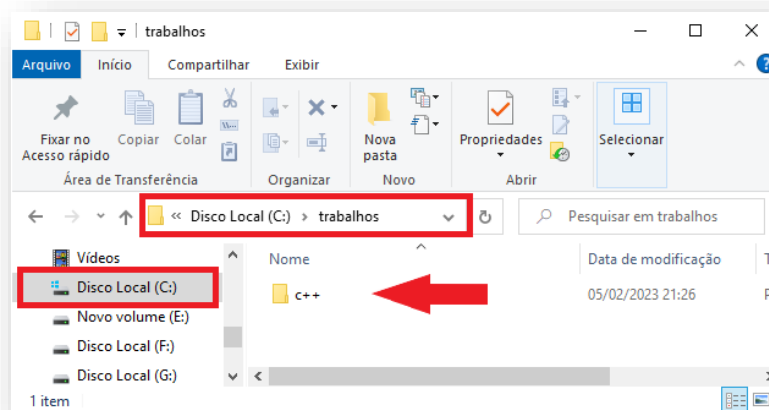
*Caso haja algum erro, no qual o sistema operacional não reconheça, tente repetir os passos anteriores, e verifique se o caminho da pasta bin do **mingw64** está correto.*

### 1.1.1.1. Configurar o Visual Studio Code para programação em C/C++

Após baixado o **Visual Studio Code**, no site <https://code.visualstudio.com/download>, crie uma pasta, no local de sua preferência, podendo ser até na **unidade C: (Disco Local (C:))**. No meu caso, criei uma pasta chamada trabalhos. E dentro de trabalhos, criei uma pasta c++. Veja as imagens a seguir da **Figura 32** e da **Figura 33**:

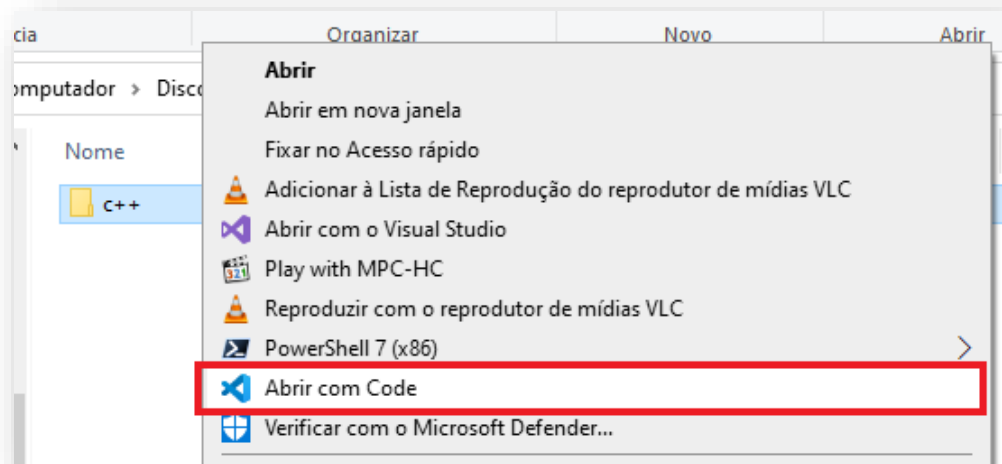


*Figura 27. Criar uma pasta chamada trabalhos. Neste caso, foi no Disco Local (C:).*



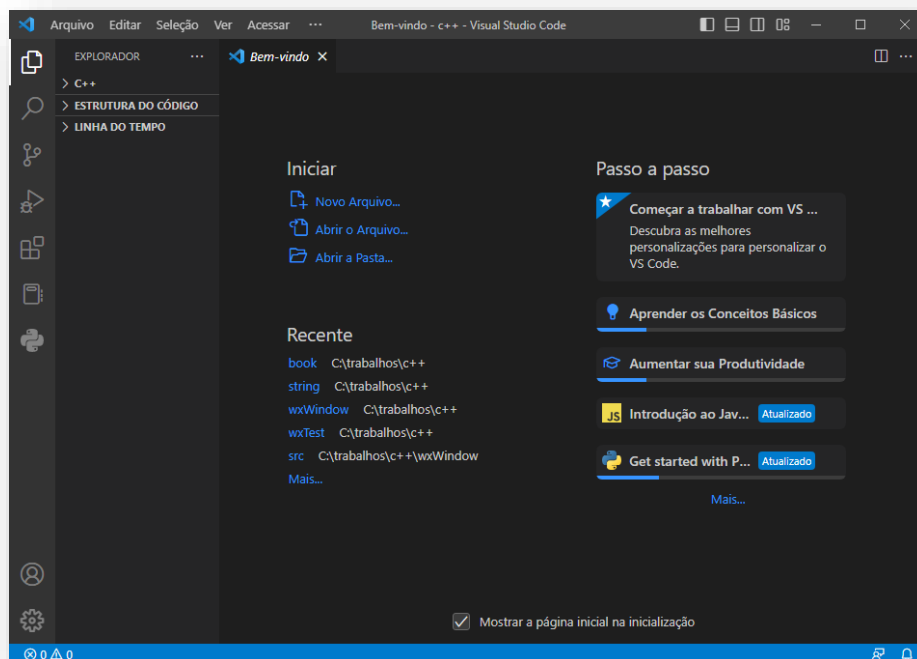
*Figura 28. A pasta c++ foi criada dentro da pasta trabalhos, que está localizada em Disco Local (C:).*

Com o botão direito do mouse sobre a pasta **c++**, vá em **Abrir com Code**. Veja a **Figura 34**.



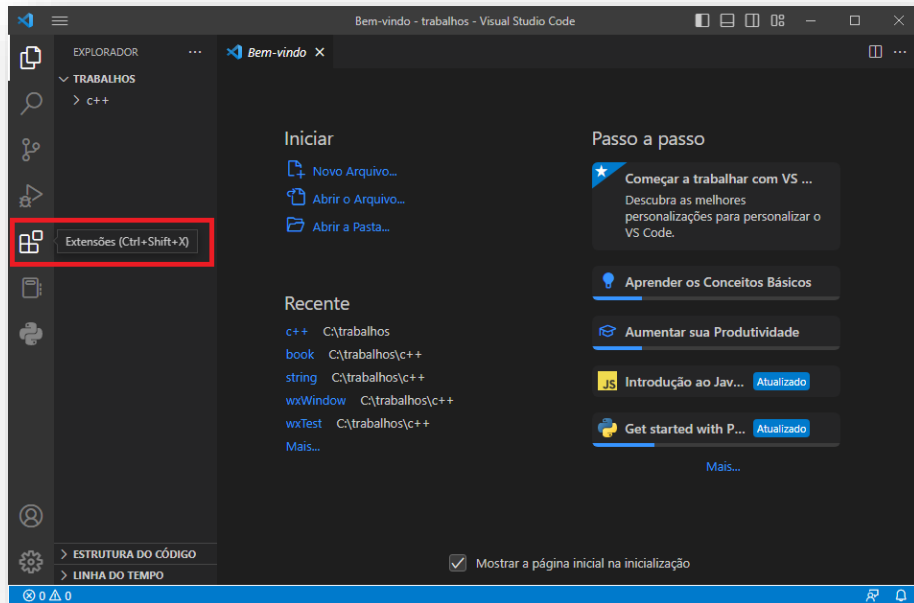
*Figura 29. Com o botão direito do mouse sobre a pasta c++, vá em Abrir com Code.*

Aparecerá, em seguida, a janela do programa do **Visual Studio Code**, com uma aba de bem-vindo. Veja a **Figura 35**.



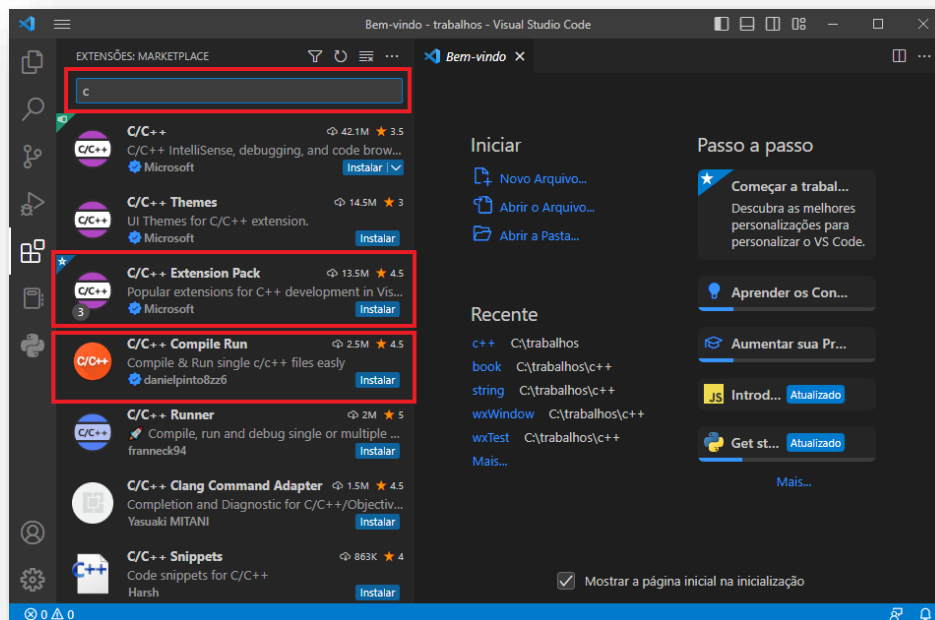
*Figura 30. Programa do Visual Studio Code, com uma aba de bem-vindo.*

Na **Figura 36**, vá em **Extensões** (ou pressione as teclas **Ctrl + Shift + X**)

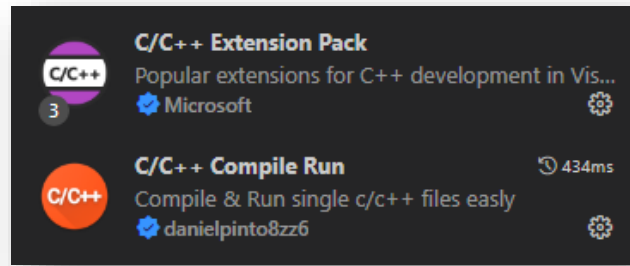


*Figura 31. Extensões.*

Digite na barra de pesquisa apenas a letra **c** ou **c++**, e dê um **Enter** para procurar. Em seguida, instale as extensões **C/C++ Extension Pack** e **C/C++ Compile Run**. Veja a **figura 37**.



*Figura 32. Instalar o C/C++ Extension Pack e o C/C++ Compile Run.*



*Figura 33. o C/C++ Extension Pack e o C/C++ Compile Run.*

### 1.1.1.2. A coleção de compiladores GNU e o projeto LLVM

O Projeto GNU, datado de 1987, é uma colaboração em massa que desenvolveu um grande conjunto de softwares livres do tipo **Unix**. O **GNU** significa “GNU’s Not Unix” – (“GNU não é Unix”). Um de seus produtos é o **GNU Compiler Collection**, ou **GCC**, que inclui o compilador GCC do C. O **GCC** está em constante desenvolvimento, guiado por um comitê diretor, e seu compilador C acompanha de perto as mudanças nos padrões C. Versões do GCC estão disponíveis para uma ampla variedade de plataformas de hardware e sistemas operacionais, incluindo **Unix**, **Linux** e **Windows**. O compilador GCC do C pode ser invocado com o comando **gcc**. E muitos sistemas que usam **gcc** farão de **cc** um alias para **gcc**.

*Na linguagem de programação, "alias" é um termo que se refere a um nome alternativo dado a uma função, comando, variável ou qualquer outra entidade programável. Esse nome alternativo pode ser usado como um atalho conveniente para se referir à entidade original.*

*No contexto do sistema operacional Unix/Linux e do Windows, um alias pode ser definido usando o comando "alias" no shell (a interface de linha de comando). Por exemplo, você pode definir um alias para um comando longo e complexo para facilitar o seu uso.*

*No caso específico que mencionamos acima, "cc" e "gcc" são ambos compiladores de C. Veja também que "cc" é geralmente associado ao compilador C padrão em sistemas Unix e Linux, enquanto "gcc" é o **GNU Compiler Collection**, que suporta várias linguagens, incluindo C. Em muitos sistemas Unix/Linux, "gcc" é, de fato, um alias para "cc", o que significa que eles são essencialmente o mesmo compilador. Isso é feito para manter a compatibilidade com scripts e códigos que usam "cc" como um comando para invocar o compilador C.*

*Portanto, no contexto específico de compiladores "cc" e "gcc", sim, em muitos sistemas, "gcc" é um alias para "cc". No entanto, isso pode variar dependendo da configuração do sistema.*

O Projeto **LLVM** fornece uma segunda substituição para o cc. O projeto é uma coleção de software relacionado a compiladores de código aberto, que remonta a um projeto de pesquisa desde o ano 2000 na Universidade de Illinois. Seu compilador **Clang** processa código C e pode ser invocado como **clang**.

Disponível em várias plataformas, incluindo **Linux**, o **Clang** se tornou o compilador C padrão para o **FreeBSD** no final do ano de 2012. Assim como o **GCC**, o **Clang** segue bastante bem o padrão C.

Em outras palavras, O **LLVM** (Low Level Virtual Machine) nada mais é que um framework de infraestrutura de compiladores projetado para **otimização, compilação e execução** de programas. Ele não é exatamente um ambiente virtual, mas sim uma **coleção de tecnologias** que permitem a **construção de compiladores, otimizadores** e outras ferramentas relacionadas à manipulação de código fonte e máquina.

O **LLVM** inclui uma série de componentes, como um conjunto de ferramentas para análise e transformação de código, além de um conjunto de bibliotecas modulares e reutilizáveis. O principal componente é o **LLVM Intermediate Representation (IR)**, uma representação intermediária de código fonte que permite que as otimizações sejam aplicadas antes da geração final de código de máquina.

O **Clang** já citado, é um compilador C/C++ de alto desempenho que utiliza o **LLVM** como **backend** para geração de código.

Observe que **Clang** é frequentemente usado como uma **alternativa** ao **GCC (GNU Compiler Collection)** e é conhecido por sua rápida compilação, mensagens de erro amigáveis e conformidade com os padrões de linguagem.

Portanto, o **LLVM** é uma infraestrutura poderosa e flexível para desenvolvimento de compiladores e ferramentas relacionadas, mas não é um ambiente virtual no sentido tradicional. Ele fornece um conjunto de tecnologias que facilitam a implementação de compiladores e otimizadores para uma variedade de linguagens de programação e plataformas de hardware.

Ambos aceitam uma opção **-v** para informações de versão, então em sistemas que usam o alias **cc** para o comando **gcc** ou **clang**, a combinação

### **cc -v**

mostra qual compilador e qual versão você está usando.

Tanto os comandos **gcc** quanto **clang**, dependendo da versão, podem exigir opções em tempo de execução para invocar padrões C mais recentes:

**gcc -std=c99 inform.c**

**gcc -std=c1x inform.c**

**gcc -std=c11 inform.c**

O primeiro exemplo invoca o padrão **C99**; o segundo invoca o esboço do padrão **C11** para versões do **GCC** anteriores à aceitação do padrão; e o terceiro invoca o padrão **C11** para versões do **GCC** que seguiram a aceitação. O compilador **Clang** usa as mesmas flags.

## Sistemas Linux

Linux é um sistema operacional popular de código aberto, semelhante ao Unix, que roda em uma variedade de plataformas, incluindo PCs e Macs. A preparação de programas C no Linux é praticamente a mesma que para sistemas Unix, exceto que você usaria o

compilador C de domínio público GCC fornecido pelo GNU. O comando de compilação fica assim:

## gcc informar.c

Observe que a instalação do GCC pode ser opcional ao instalar o Linux, então você (ou alguém) pode ter que instalar o GCC se ele não tiver sido instalado anteriormente. Normalmente, a instalação torna `cc` um alias para `gcc`, então você pode usar `cc` na linha de comando em vez de `gcc` se desejar. Você pode obter mais informações sobre o GCC, incluindo informações sobre novos lançamentos em:

<https://www.gnu.org/software/gcc/index.html>

## Compiladores de linha de comando para PC

Os compiladores C não fazem parte do pacote padrão do Windows, portanto, talvez seja necessário obter e instalar um compilador C. **Cygwin** e **MinGW** são downloads gratuitos que disponibilizam o compilador **GCC** para uso de linha de comando em um PC. O **Cygwin** é executado em sua própria janela, que tem uma aparência de prompt de comando, mas que imita um ambiente de linha de comando do Linux. O **MinGW**, por outro lado, é executado no modo Prompt de Comando do Windows. Eles vêm com a versão mais recente (ou quase mais recente) do GCC, que suporta C99 e pelo menos parte do C11. O Borland C++ Compiler 5.5 é outro download gratuito; ele suporta C90.

Os arquivos de código-fonte devem ser arquivos de texto, não arquivos de processador de texto. (Os arquivos do processador de texto contêm muitas informações adicionais sobre fontes e formatação.) Você deve usar um editor de texto, como o [Visual Studio Code](#) da Microsoft. Você pode usar qualquer processador de texto se usar o recurso **Salvar** como para salvar o arquivo em modo de texto. O arquivo deve ter extensão `“.c”`. Alguns processadores de texto adicionam automaticamente uma extensão `“.txt”` aos arquivos de texto. Se isso acontecer com você, será necessário alterar o nome do arquivo, substituindo `“txt”` por `“c”`.

Compiladores C para PC normalmente, mas nem sempre, produzem arquivos de código-objeto intermediários com uma extensão `“.obj”`. Ao contrário dos compiladores Unix, esses compiladores normalmente não removem esses arquivos quando terminam. Alguns compiladores produzem arquivos em linguagem assembly com extensões `“.asm”` ou usam algum formato especial próprio.

Alguns compiladores executam o vinculador automaticamente após a compilação; outros podem exigir que você execute o vinculador manualmente. A vinculação resulta no arquivo executável, que anexa a extensão `.EXE` ao nome base do código-fonte original. Por exemplo, compilar e vincular um arquivo de código-fonte chamado **program.c** produz um arquivo chamado **program.exe**. Você pode executar o programa digitando o nome base na linha de comando:

**C:\>program**

## Ambientes de desenvolvimento integrados – IDE (Windows)

Vários fornecedores, incluindo **Microsoft**, **Embarcadero** e **Digital Mars**, oferecem ambientes de desenvolvimento integrados baseados em Windows, ou **IDEs**.

Atualmente, a maioria são compiladores C e C++ combinados. Os downloads gratuitos incluem **Microsoft Visual Studio Express** e **Pelles C**. Todos possuem ambientes rápidos e integrados para montar programas C. O ponto principal é que cada um desses programas possui um editor integrado que você pode usar para escrever um programa em C. Cada um fornece menus que permitem nomear e salvar seu arquivo de código-fonte, bem como menus que permitem compilar e executar seu programa sem sair da IDE. Cada um retorna você no editor se o compilador encontrar algum erro, e cada um identifica as linhas incorretas e as combina com as mensagens de erro apropriadas.

As IDEs do Windows podem ser um pouco intimidadores no início porque oferecem uma variedade de alvos (targets) — ou seja, uma variedade de ambientes nos quais o programa será usado. Por exemplo, eles podem oferecer a você a opção de programas do **Windows de 32 bits**, programas do **Windows de 64 bits**, arquivos de biblioteca de vínculo dinâmico (**DLLs**) e assim por diante. Muitos dos objetivos envolvem trazer suporte para a interface gráfica do Windows. Para gerenciar essas (e outras) opções, normalmente você cria um projeto ao qual adiciona os nomes dos arquivos de código-fonte que usará. As etapas precisas dependem do produto que você usa. Normalmente, você usa primeiro o menu **Arquivo** ou o menu **Projeto** para criar um projeto. O importante é escolher a forma correta de projeto. Os exemplos neste livro são exemplos genéricos projetados para serem executados em um ambiente simples de linha de comando. As várias IDEs do Windows oferecem uma ou mais opções para atender a essa suposição pouco exigente. O **Microsoft Visual Studio**, por exemplo, oferece a opção **Win32 Console Application**. Para outros sistemas, procure uma opção usando termos como **DOS EXE**, **Console** ou **Terminal** em modo de caractere. Esses modos executarão seu programa executável em uma janela semelhante a um console. Depois de ter o tipo de projeto correto, use o menu IDE para abrir um novo arquivo de código-fonte. Para a maioria dos produtos, você pode fazer isso usando o menu Arquivo. Talvez seja necessário executar etapas adicionais para adicionar o arquivo de origem ao projeto.

Como as IDEs do Windows normalmente lidam com C e C++, você precisa indicar que deseja um programa C. Com alguns aplicativos, você usa o tipo de projeto para indicar que deseja usar a linguagem C. Com outros, como o **Microsoft Visual C++**, você usa a extensão de arquivo **.c** para indicar que deseja usar C em vez de C++. No entanto, a maioria dos programas C também funciona como programas C++.

Um problema que você pode encontrar é que a janela que mostra a execução do programa desaparece quando o programa termina. Se for esse o seu caso, você pode pausar o programa até pressionar a tecla **Enter**. Para fazer isso, adicione a seguinte linha ao final do programa, logo antes da instrução **return**:

```
getchar();
```



Esta linha lê um pressionamento de tecla, então o programa fará uma pausa até que você pressione a tecla **Enter** novamente. Às vezes, dependendo de como o programa funciona, já pode haver um pressionamento de tecla em espera. Nesse caso, você terá que usar `getchar()` duas vezes:

**`getchar();`**

**`getchar();`**

Por exemplo, se a última coisa que o programa fez foi pedir para você inserir seu peso, você teria digitado seu peso e pressionado a tecla **Enter** para inserir os dados. O programa leria o peso, o primeiro **`getchar()`** leria a tecla **Enter** e o segundo **`getchar()`** faria o programa pausar até que você pressionasse **Enter** novamente. Se isso não faz muito sentido para você agora, fará depois que você aprender mais sobre entradas (inputs) em C. E vamos lembrá-lo mais tarde sobre essa abordagem.

Embora as várias IDEs tenham muitos princípios gerais em comum, os detalhes variam de aplicativo para aplicativo e, dentro de uma linha de softwares, de versão para versão. Você terá que fazer algumas experiências para aprender como seu compilador funciona. Talvez você até precise ler o manual ou tentar um tutorial online.

# Capítulo 2

## 2.1. Um exemplo simples de C

Vamos dar uma olhada em um programa C simples. Este programa, mostrado no **Código 2.1**, serve para apontar alguns dos recursos básicos da programação em C. Antes de ler a próxima explicação linha por linha do programa, leia o **Código 2.1** para ver se você consegue descobrir por si mesmo o que isso fará.

```
#include <stdio.h>    /* Inclui o arquivo de cabeçalho stdio.h */

int main (void)      /* Função principal que inicia a aplicação */
{
    int num;          /* Declara uma variável do tipo inteiro */

    num = 1;          /* Atribui um valor à variável num */

    /* Usa a função printf() para imprimir mensagens no Terminal */
    printf("Eu sou um simples ");
    printf("computador.\n");
    printf("Meu numero favorito eh %d porque ele eh o primeiro.\n", num);

    return 0;
}
```

Se você acha que este programa irá imprimir algo na sua tela, você está certo! Exatamente o que será impresso pode não ser aparente, então execute o programa e veja os resultados. Primeiro, vá no [Visual Code Studio](#) ou use seu editor favorito do seu compilador para criar um arquivo contendo o texto do **Código 2.1**. Dê ao arquivo um nome que termine em “.c” e que atenda aos requisitos de nome do seu sistema local. Você pode usar **cod21.c**, por exemplo. Agora compile. Veja o comando no terminal:

```
gcc -std=c11 cod21.c -o program.exe
```

ou

```
gcc -g cod21.c -o program.exe
```

Para executar o arquivo **program.exe** no terminal, se for no prompt comando do Windows, basta digitar **program**. Mas se você estiver usando o **Visual Studio Code**, que integra o **Power Shell do Windows**, para ter uma experiência de sistemas **Unix-Like**, apenas digite o seguinte comando:

```
./program.exe
```

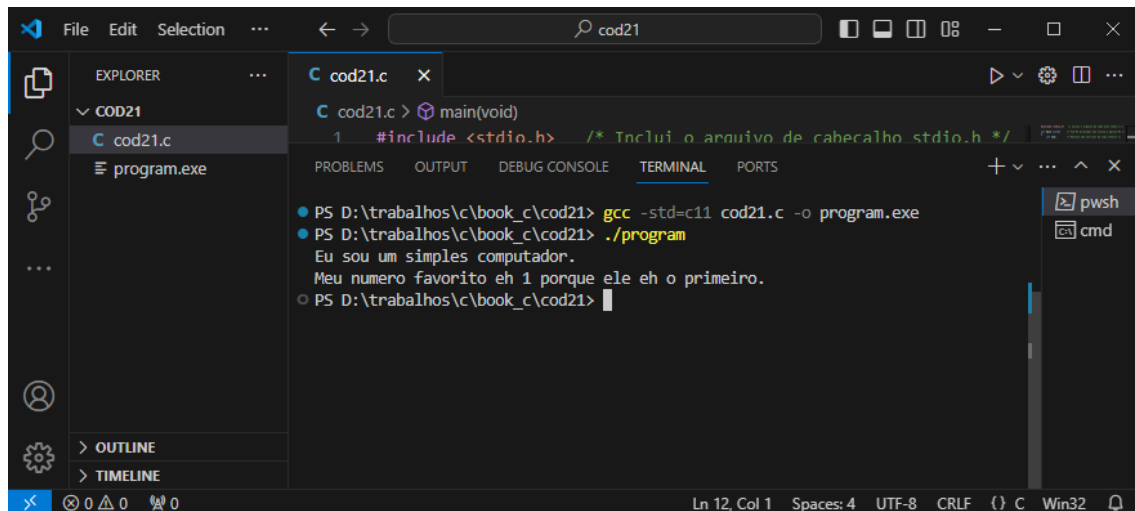
ou

`./program`

Veja que no segundo caso não foi necessário colocar a extensão “`.exe`”.

Veja o resultado quando invocamos o executável pelo terminal:

```
Eu sou um simples computador.  
Meu numero favorito eh 1 porque ele eh o primeiro.
```



```
PS D:\trabalhos\c\book_c\cod21> gcc -std=c11 cod21.c -o program.exe  
PS D:\trabalhos\c\book_c\cod21> ./program  
Eu sou um simples computador.  
Meu numero favorito eh 1 porque ele eh o primeiro.  
PS D:\trabalhos\c\book_c\cod21>
```

Figura 34. Terminal do Power Shell do Visual Studio Code.

Resumindo, esse resultado não é muito surpreendente, mas o que aconteceu com os “`\n`”, e com o “`%d`” no programa? E algumas linhas do programa parecem estranhas! É hora de uma boa explicação.

Ajustando o programa, caso você utilize uma IDE, quando executado o programa, através de um botão “run”, e venha aparecer uma janela de Terminal do Prompt Comando e depois desapareça.

A saída deste programa abriu um terminal brevemente na tela e depois desapareceu? Alguns ambientes de desenvolvimento executam o programa em uma janela separada e fecham automaticamente quando o programa termina. Nesse caso, você pode fornecer um código extra para fazer com que a janela permaneça aberta até você pressionar uma tecla. Uma maneira é adicionar a seguinte linha antes da instrução **return**:

**getchar();**

Este código faz com que o programa aguarde um pressionamento de tecla, de forma que a janela permaneça aberta até que você pressione, por exemplo, o **Enter**.

```
#include <stdio.h> /* Inclui o arquivo de cabeçalho stdio.h */
```

```
int main (void)    /* Função principal que inicia a aplicação */
{
    int num;        /* Declara uma variável do tipo inteiro */

    num = 1;        /* Atribui um valor à variável numm */

    /* Usa a função printf() para imprimir mensagens no Terminal */
    printf("Eu sou um simples ");
    printf("computador.\n");
    printf("Meu numero favorito eh %d porque ele eh o primeiro.\n", num);

    getchar();

    return 0;
}
```

## O exemplo explicado

Faremos duas passagens pelo código-fonte do programa. A primeira passagem (“Passagem 1: Sinopse Completa”) destaca o significado de cada linha para ajudá-lo a ter uma ideia geral do que está acontecendo. A segunda etapa (“Passagem 2: Detalhes do Programa”) explora implicações e detalhes específicos para ajudá-lo a obter uma compreensão mais profunda. A Figura 2.1 resume as partes de um programa C; inclui mais elementos do que nosso primeiro exemplo usa.

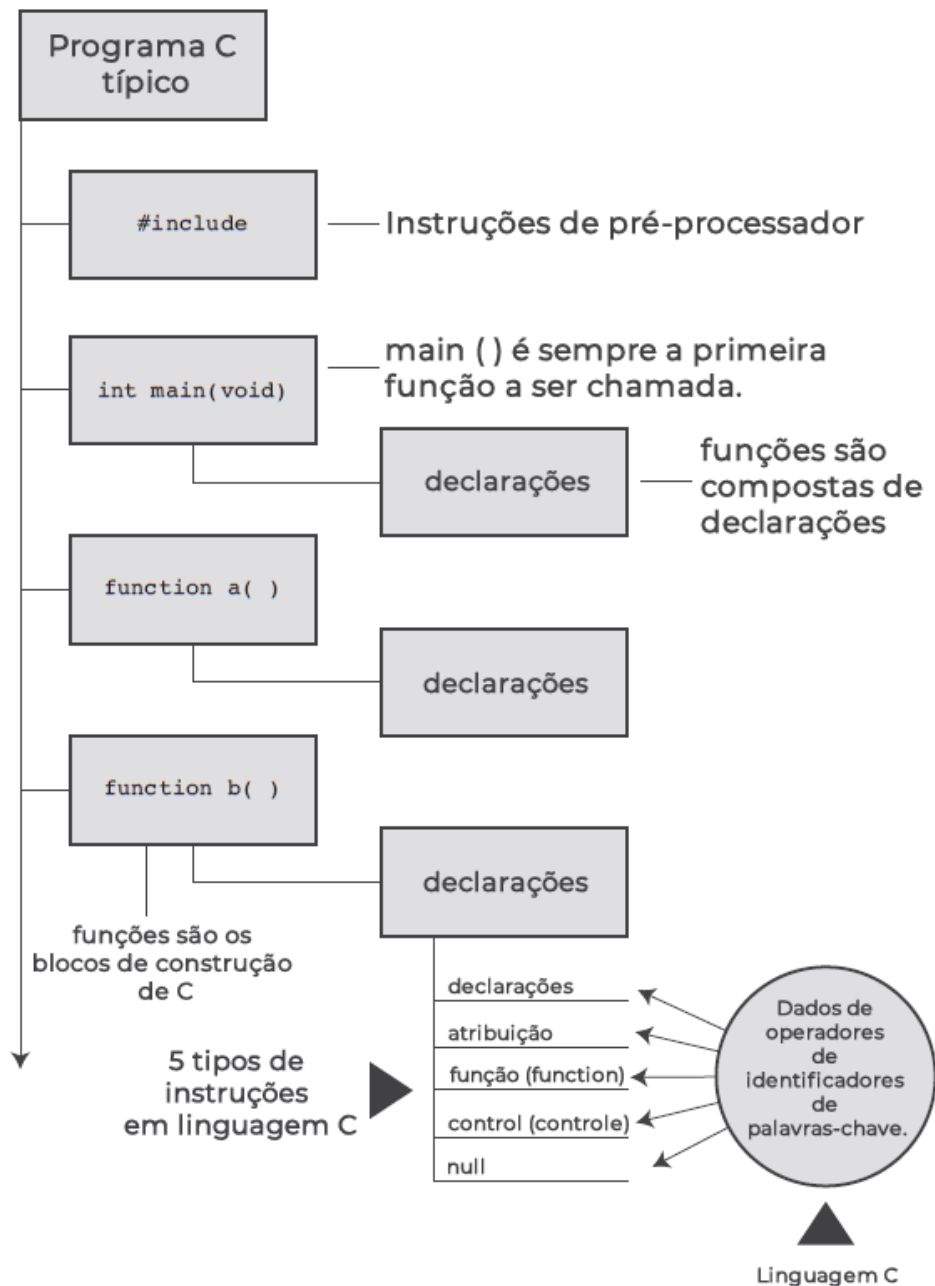


Figura 35. Anatomia de um programa em C.

## Passo 1: Sinopse Completa

Esta seção apresenta cada linha do programa seguida de uma breve descrição; a próxima seção (Passo 2) explora mais detalhadamente os tópicos levantados aqui.

### #include <stdio.h>

O comando acima seria o mesmo que incluir outro arquivo. Ou seja, esta linha diz ao compilador para incluir as informações encontradas no arquivo **stdio.h**, que é uma parte padrão de todos os pacotes do compilador C; este arquivo fornece suporte para entrada de teclado (**input**) e exibição de saída (**output**).

A diretiva **#** (hashtag) significa pré-processador! Ou melhor, “**#**” é uma diretiva de pré-processador que tem a função de realizar algo antes do pré-processamento de compilação! Esse fazer algo é o comando **include**, que traduzindo, seria um comando para incluir algo entre o sinal de menor (<) e sinal de maior (>).

Veja que **std** significa **standard**, que em português é o mesmo que **padrão**. O caractere “**i**” tem o significado de **input** (entrada); e “**o**”, de **output** (saída). Então o arquivo de cabeçalho **stdio.h** seria um arquivo de entrada e saída padrão. A extensão “**h**” significa **header** (cabeçalho), por isso arquivo de cabeçalho.

Os programas C consistem em uma ou mais funções, os módulos básicos de um programa C. Este programa consiste em uma função chamada **main**. Os parênteses identificam **main()** como o nome de uma função. O **int** indica que a função **main()** retorna um **número inteiro** e o **void** indica que **main()** não aceita **nenhum argumento**. Esses são assuntos que abordaremos mais tarde. No momento, basta aceitar **int** e **void** como parte da maneira padrão **ANSI C** para definir **main()**.

**OBS:** Se você tiver um compilador C pré-ANSI, omita **void**; você pode querer obter algo mais recente para evitar incompatibilidades.

### Como fazer comentários para documentação: /\* um programa simples \*/

Os símbolos /\* e \*/ incluem comentários – observações que ajudam a esclarecer um programa. Eles são destinados apenas ao leitor e são ignorados pelo compilador.

/\* → Abre o comentário.

\*/ → Fecha o comentário.

### Início do corpo da função, que seria o abre chaves: {

Esta chave de abertura marca o início das instruções que compõem a função. Uma chave de fechamento ( **}** ) marca o fim da definição da função.

### Declarando o nome de uma variável e seu tipo: int num;

**int** num;      // variável com nome “num” do tipo inteiro

Esta instrução anuncia que você está usando uma variável chamada **num** e que num será um **int** (**inteiro** – em inglês seria **integer**).

Fazendo uma atribuição de um valor inteiro a uma variável, no caso, “num” que recebe o valor inteiro 1.

```
num = 1;
```

A instrução `num = 1;` atribui o valor 1 à variável chamada num.

**OBS:** O sinal de igual “=” não significa que **num** é igual a **1**! Este sinal simboliza uma atribuição. O valor está sendo atribuído a uma **variável**, mas nunca o local deste armazenamento (**variável**) será igual ao valor atribuído, porque, na maioria dos casos, as variáveis podem **variar** de valores! Em outras palavras, as variáveis são locais ou posições nas memórias do computador que guardam um valor de cada vez! Se o número 1 está guardado na variável num, e depois você queira colocar o número 10, então sairá o número 1 e ficará guardado o número 10.

Uma instrução de chamada de função: `printf("Eu sou um simples");`

A primeira instrução usando **printf()** exibe a frase “**Eu sou um simples**” na tela, deixando o cursor na mesma linha. Aqui **printf()** faz parte da biblioteca C padrão. É denominado **função**. E usar uma função no programa é denominado **chamada de função**.

Outra instrução de chamada de função: `printf("computador.\n");`

A próxima chamada para a função **printf()** segue no computador até o final da última frase impressa. O `\n` é um código que informa ao computador para iniciar uma nova linha – ou seja, para mover o cursor para o início da próxima linha.

Observe, porém, que os caracteres “`\n`” não são exibidos na tela. A barra invertida (`\`) é chamada de **caractere de escape**. Ela indica que a função **printf** deve fazer algo **fora do esperado**. Quando uma barra invertida “`\`” é encontrada em uma string de caracteres, o compilador examina o próximo caractere e o combina com a barra invertida para formar uma **sequência de escape**. Portanto, esta sequência de escape “`\n`” significa **nova linha (newline)**. Quando uma sequência de nova linha aparece na string de saída, através de uma função **printf**, a nova linha faz com que o cursor se posicione no início da próxima linha na tela. Algumas sequências de escape comuns são listadas na tabela seguinte.

Sequência de escape	Descrição
<code>\n</code>	Nova linha. Posiciona o cursor da tela no início da próxima linha.
<code>\t</code>	Tabulação horizontal. Move o cursor da tela para a próxima posição de tabulação.
<code>\a</code>	Alerta. Faz soar o alarme do sistema.
<code>\\</code>	Barra invertida. Insere um caractere de barra invertida em uma string.
<code>\"</code>	Aspas. Insere um caractere de aspas em uma string.

*Algumas sequências comuns de escape.*

```
printf("Meu número favorito é %d porque é o primeiro.\n", num);
```

O último uso da função **printf()** imprime o valor de **num** (que é 1) incorporado na frase entre aspas. O **%d** instrui o computador, onde e de que forma, imprimir o valor de **num**. Esse **%d** significa que é um especificador de conversão, indicando que um inteiro será exibido. O “**d**” indica que é um número decimal do tipo inteiro. O caractere “**%**” é tratado como um caractere especial, que inicia um especificador de conversão. Neste caso, seria um especificador para um número inteiro decimal (**%d**). Ele não será imprimido no terminal, mas apenas será substituído pelo valor da variável que é o número inteiro 1. Ele apenas vai indicar que será substituído pelo próximo argumento. No exemplo acima, o argumento correspondente seria o “**num**” após a vírgula.

## Uma declaração de retorno para a função do tipo inteiro: **return 0;**

```
return 0;
```

Uma função C pode fornecer ou retornar um número para o agente que o utilizou. Simplesmente irá indicar que o programa concluiu com sucesso! Por enquanto, considere esta linha como o fechamento apropriado para uma função **main()**.

## Fim da função **main(): }**

Como prometido, o programa termina com chave de fechamento.

## *Passo 2: Detalhes do Programa*

Agora que você tem uma visão geral do Código 2.1, daremos uma olhada mais de perto. Mais uma vez, examinaremos as linhas individuais do programa, desta vez usando cada linha de código como ponto de partida para aprofundar os detalhes por trás do código e como base para desenvolver uma perspectiva mais geral dos recursos de programação em C.

## #include Diretivas e Arquivos de Cabeçalho

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

Esta é a linha que inicia o programa. O efeito de **#include <stdio.h>** é o mesmo que se você tivesse digitado todo o conteúdo do arquivo **stdio.h** em seu arquivo no ponto onde a linha **#include** aparece. Na verdade, é uma operação de copiar/recortar e colar. Os arquivos da pasta **include** fornecem uma maneira conveniente de compartilhar informações comuns a muitos programas.

A instrução **#include** é um exemplo de **diretiva de pré-processador** C. Em geral, os compiladores C realizam algum **trabalho preparatório no código-fonte** antes de compilar; isso é denominado **pré-processamento**.

O arquivo **stdio.h** é fornecido como parte de todos os pacotes do compilador C. Ele contém informações sobre funções de entrada (input) e saída (output), como **printf()**, para uso do compilador. O nome significa **cabeçalho de entrada/saída padrão** - (stands for **standard input/output header – std + i (input) + o (output) .h (header)**). As pessoas, que programam na linguagem C, chamam de cabeçalho uma coleção de informações



que vai no topo de um arquivo, e suas implementações geralmente vêm com vários arquivos de cabeçalho.

Na maior parte, os **arquivos de cabeçalho** contêm informações usadas pelo compilador para construir o programa **executável final**. Por exemplo, eles podem definir constantes ou indicar os nomes das funções e como devem ser utilizadas. Mas o código real de uma função está em um **arquivo de biblioteca de código pré-compilado**, não em um arquivo de cabeçalho. O **componente vinculador** (linker) do compilador se encarrega de encontrar o **código da biblioteca** que você precisa. Resumindo, os arquivos de cabeçalho ajudam a orientar o compilador na montagem correta do seu programa.

A **ANSI/ISO do C** padronizou quais arquivos de cabeçalho um compilador C deve disponibilizar. Alguns programas precisam incluir **stdio.h** e outros não. A documentação para uma implementação C específica deve incluir uma descrição das funções da biblioteca C. Estas descrições de funções identificam quais arquivos de cabeçalho são necessários. Por exemplo, a descrição de **printf()** diz para usar **stdio.h**. A omissão do arquivo de cabeçalho adequado pode não afetar um programa específico, mas tenha essa confiança. Cada vez que esta explicação usar funções de biblioteca, ele usará os arquivos de inclusão especificados pelo padrão **ANSI/ISO** para essas funções.

### **Observe por que a entrada e a saída não estão integradas**

*Talvez você esteja se perguntando por que recursos tão básicos como entrada (input) e saída (output) **não** são incluídos automaticamente. Uma resposta é que nem todos os programas usam esse pacote de **E/S** (entrada/saída) - I/O (input/output), e parte da filosofia de **C** é evitar carregar **peso desnecessário**. Este princípio de uso econômico de recursos torna **C** popular para **programação embarcada** - por exemplo, escrever código para um chip que controla um sistema de combustível automotivo ou um reproduutor de Blu-ray. Aliás, a linha **#include** nem sequer é uma declaração da linguagem **C**! O símbolo **#** na linha 1 identifica o local como deve ser manipulado pelo **pré-processador C** antes que o compilador assuma o controle. Você encontrará mais exemplos de instruções de pré-processador posteriormente!*

## A função principal **main()**

### **int main(void)**

A próxima linha do programa proclama uma função chamada **main**. É verdade que **main** é um nome bastante simples, mas é a única opção disponível. Um programa C (com algumas exceções com as quais não nos preocuparemos) sempre começa a execução com a função chamada **main()**. Você é livre para escolher nomes para outras funções que usar, mas **main()** deve estar lá para iniciar as coisas. E os parênteses? Eles identificam **main()** como uma função. Você aprenderá mais sobre funções em breve. Por enquanto, lembre-se de que funções são os módulos básicos de um programa C.

O **int** é o tipo de retorno da função **main()**. Isso significa que o tipo de valor que **main()** pode retornar é um número **inteiro**. Voltar para onde? Para o sistema operacional – depois voltaremos a esta questão!

Os parênteses após o nome de uma função geralmente incluem informações que são transmitidas à função. Para este exemplo simples, nada está sendo repassado, então os parênteses contêm a palavra **void** (vazio). (“Sequências de caracteres e funções de string”, apresenta um segundo formato que permite que informações sejam passadas para **main()** do sistema operacional.)

Se você navegar pelo código C antigo, verá frequentemente programas começando com o seguinte formato:

### ***main()***

O padrão C90 tolerou esta forma de má vontade, mas os padrões **C99** e **C11** não! Portanto, mesmo que seu compilador atual permita fazer isso, não faça isso!

A seguir veja uma outra maneira de escrever:

### ***void main()***

**Mas cuidado!** Alguns compiladores permitem isso, mas nenhum dos padrões jamais listou isso como uma opção reconhecida ou opcional. Portanto, os compiladores não precisam aceitar esta forma, e vários não o reconhecem!. Novamente, siga o formato padrão e você não terá problemas se mover um programa de um compilador para outro.

## Comentários

```
/* um programa simples */
```

As partes do programa entre os símbolos ***/\* \*/*** são comentários. O uso de comentários torna mais fácil para alguém (inclusive você) entender seu programa. Uma característica interessante dos comentários em C é que eles podem ser colocados em qualquer lugar, até mesmo na mesma linha do material que explicam. Um comentário mais longo pode ser colocado em sua própria linha ou até mesmo espalhado por mais de uma linha. Tudo entre a abertura ***/\**** e o fechamento ***\*/*** é ignorado pelo compilador. A seguir estão alguns formulários de comentários válidos e inválidos:

```
/* Este é um comentário C. */

/* Este comentário, sendo um tanto prolixo, está espalhado por
duas linhas. */

/*
Você também pode fazer isso.
*/

/* Mas isso é inválido porque não há marcador final.
```

C99 adicionou um segundo estilo de comentários, popularizado por **C++** e **Java**. O novo estilo usa os símbolos `//` para criar comentários confinados a uma única linha:

```
// Aqui está um comentário confinado a uma linha.  
int variavel; // Esses comentários também podem ir aqui.
```

Como o final da linha marca o fim do comentário, esse estilo precisa de marcadores de comentário apenas no início do comentário.

O arquivo mais recente é uma resposta a um problema potencial com o arquivo antigo. Suponha que você tenha o seguinte código:

```
/*  
Eu espero que isso funcione.  
*/  
x = 100;  
y = 200;  
/* Agora, outra coisa. */
```

Em seguida, suponha que você decida remover a quarta linha e excluir acidentalmente a terceira linha (o `*/`) também. O código então se torna:

```
/*  
Eu espero que isso funcione.  
y = 200;  
/* Agora, outra coisa. */
```

Agora o compilador emparelha `/*` na primeira linha com `*/` na quarta linha, transformando todas as quatro linhas em um comentário, incluindo a linha que deveria fazer parte do código. Como o formulário `//` não se estende por mais de uma linha, ele não pode levar a esse problema de “**desaparecimento** de código”.

Alguns compiladores podem não suportar esse recurso; outros podem exigir a alteração de uma configuração do compilador para ativar os recursos C99 ou C11.

Esta explicação, baseada na teoria de que consistência desnecessária pode ser enfadonha, utiliza ambos os tipos de comentários.

## Chaves, corpos e blocos

```
{  
...  
}
```

Na **Código 2.1**, as chaves “{ .. }” delimitam a função `main()`. Em geral, todas as funções C usam chaves para marcar o início e também o fim do **corpo de uma função**. A presença deles é obrigatória, por isso não os deixe de fora. Somente chaves ( { } ) funcionam para essa finalidade, não parênteses ( ( ) ) e nem colchetes ( [ ] ).

Chaves também podem ser usadas para reunir instruções dentro de uma função em uma unidade ou bloco. Se você estiver familiarizado com **Pascal**, **ADA**, **Modula-2** ou **Algol**, reconhecerá os colchetes como sendo semelhantes no início e no fim nessas linguagens.

## Declarações

```
int num;
```

Esta linha do programa é chamada de **instrução de declaração**. A instrução de declaração é um dos recursos mais importantes de C. Este exemplo específico declara duas coisas. Primeiro, em algum lugar da função, você tem uma **variável** chamada **num**. Segundo, o **int** proclama **num** como um **número inteiro** – ou seja, um número sem ponto decimal ou parte fracionária. (**int** é um exemplo de **tipo de dados**.) O compilador usa essas informações para **organizar espaço de armazenamento adequado na memória** para a **variável num**. O **ponto-e-vírgula** no final da linha **identifica a linha** como uma **instrução em C**. O **ponto-e-vírgula** faz parte da instrução! Ele não é apenas um separador entre instruções como em Pascal.

A palavra **int** é uma **palavra-chave** em C que identifica um dos **tipos básicos de dados de C**. As **Palavras-chave** são palavras usadas para expressar uma linguagem e você não poderá usá-las para outros fins. Por exemplo, você não pode usar **int** como o nome de uma função ou de uma variável. No entanto, essas restrições de **palavras-chave** não se aplicam fora da linguagem, portanto, não há problema em nomear um gato ou um filho favorito com a palavra **int**. Mas ficará estranho! Imagine você dizendo ao seu filho: - **Int**, dê-me aqui para o papai ou para a mamãe essa **chave**!

A palavra **num** neste exemplo é um **identificador** — isto é, um **nome** que você seleciona para **uma variável**, **uma função** ou alguma **outra entidade**. É um nome que você escolhe quando vai ter um filho! Precisa criar na memória uma entidade com um nome! Assim, a declaração conecta um determinado **identificador** a um **determinado local na memória** do computador e também estabelece o tipo de informação, ou tipo de dados, a ser armazenado naquele local.

Identificador do Espaço de memória: **num**

Espaço na memória para valor inteiro.

**4 bytes**

Em C, todas as variáveis devem ser declaradas antes de serem usadas. Isso significa que você deve fornecer listas de todas as variáveis que usa em um programa e mostrar qual é o tipo de dados de cada variável. Declarar variáveis é considerada uma boa prática de programação e, em C, é obrigatória.

Tradicionalmente, C exige que as variáveis sejam declaradas no início de um bloco, sem que nenhum outro tipo de instrução possa vir antes de qualquer uma das declarações. Ou seja, o corpo de `main()` pode ter a seguinte aparência:

```
#include <stdio.h>

int main() // regras tradicionais
{
    int doors; // declara doors como tipo inteiro
    int dogs;  // declara dogs como tipo inteiro

    doors = 5; // atribui valor inteiro 5 à variável doors
    dogs = 3;  // atribui valor inteiro 3 à variável dogs

    //outras declarações

    return 0;
}
```

C99 e C11, seguindo a prática do C++, permitem colocar declarações sobre qualquer lugar em um bloco. No entanto, você ainda deve declarar uma variável antes de seu primeiro uso. Portanto, se o seu compilador suportar esse recurso, seu código poderá ser semelhante ao seguinte:

```
#include <stdio.h>

int main() // regras tradicionais
{
    int doors; // 1º declara doors como tipo inteiro
    doors = 5; // atribui valor inteiro 5 à variável doors

    // mais declarações

    int dogs; // 2º declara dogs como tipo inteiro
    dogs = 3;  // atribui valor inteiro 3 à variável dogs

    //outras declarações

    return 0;
}
```

Para maior compatibilidade com sistemas mais antigos, esta explicação seguirá a convenção original. Neste ponto, você provavelmente tem três perguntas. Primeiro, quais são os **tipos de dados**? Em segundo lugar, que opções você tem ao **selecionar um nome**? Terceiro, por que você precisa **declarar variáveis**? Vejamos algumas respostas.

## Tipos de dados

A linguagem C lida com vários tipos (ou tipos) de dados: **inteiros**, **caracteres** e **ponto flutuante**, por exemplo. Declarar uma variável como um número inteiro ou um tipo de caractere permite que o computador **armazene**, **busque** e **interprete os dados adequadamente**. Depois vamos investigar a variedade de tipos disponíveis nas próximas explicações!

## Escolha do nome

Você deve usar nomes significativos ou intuitivos (ou identificadores) para variáveis (como **ovelha\_cont** em vez de **x3** se seu programa contar ovelhas). Tome cuidado para não criar um nome como **3x**. Nunca comece os identificadores de variáveis com números! E também não coloque espaços: **3 x**. A seguir, veremos o que pode e o que não pode!

Se o nome não for suficiente, use comentários para explicar o que as variáveis representam. Documentar um programa dessa maneira é uma das técnicas básicas de uma boa programação.

Com C99 e C11 você pode deixar o nome de um identificador tão longo quanto desejar, mas o compilador precisa considerar apenas os primeiros **63 caracteres** como significativos. Para identificadores externos apenas **31 caracteres** precisam ser reconhecidos. Este é um aumento substancial em relação ao requisito C90 de 31 caracteres e seis caracteres, respectivamente, e os compiladores C mais antigos geralmente paravam no máximo de **oito caracteres**. Na verdade, você pode usar mais do que o número máximo de caracteres, mas o compilador não é obrigado a prestar atenção aos caracteres extras. O que isto significa? Se você tiver dois identificadores com **63 caracteres cada** e **idênticos**, exceto por um caractere, o compilador deverá reconhecê-los como distintos um do outro. Se você tiver dois identificadores com **64 caracteres** e **idênticos**, exceto o caractere final, o compilador poderá reconhecê-los como distintos ou não; a norma não define o que deve acontecer nesse caso.

Os caracteres à sua disposição são **letras minúsculas**, **letras maiúsculas**, **dígitos** e **sublinhado** (**\_**). O primeiro caractere deve ser **uma letra** ou **um sublinhado**. A seguir estão alguns exemplos:

Nomes Válidos	Nomes Inválidos
<b>number</b>	<b>\$Z]**</b>
<b>cat2</b>	<b>2cat</b>
<b>Hot_Tub</b>	<b>Hot-Tub</b>
<b>taxRate</b>	<b>tax rate</b>
<b>_value</b>	<b>_ não</b>
<b>valor_2</b>	<b>valor 2</b>

Os sistemas operacionais e a biblioteca C geralmente usam identificadores com um ou dois caracteres de sublinhado iniciais, como em `_value`, portanto, é melhor evitar esse uso. Os rótulos padrão que começam com um ou dois caracteres de sublinhado, como identificadores de biblioteca, são reservados. Isso significa que embora não seja um erro de sintaxe usá-los, pode levar a conflitos de nomes.

Os nomes em C diferenciam maiúsculas de minúsculas (**case-sensitive**), o que significa que uma letra **maiúscula** é considerada distinta da letra **minúscula** correspondente. Portanto, **estrelas** são diferentes de **Estrelas** e **ESTRELAS**.

## Quatro boas razões para declarar variáveis

Algumas linguagens mais antigas, como as formas originais de FORTRAN e BASIC, permitem usar variáveis sem declará-las. Então, por que você não pode adotar essa abordagem fácil em C? Aqui estão alguns motivos:

- Colocar todas as variáveis em um só lugar torna mais fácil para o leitor entender do que se trata o programa. Isto é particularmente verdadeiro se você der nomes significativos às suas variáveis (como taxa de imposto em vez de `r`). Se o nome não for suficiente, use comentários para explicar o que as variáveis representam. Documentar um programa dessa maneira é uma das técnicas básicas de uma boa programação.
- Pensar em quais variáveis declarar incentiva você a fazer algum planejamento antes de começar a escrever um programa. De quais informações o programa precisa para começar? O que exatamente eu quero que o programa produza como saída? Qual é a melhor forma de representar os dados?
- Declarar variáveis ajuda a evitar um dos bugs mais sutis e difíceis de encontrar na programação: o nome da variável com erro ortográfico. Por exemplo, suponha que em alguma linguagem que não possui declarações, você fez a declaração

**RAIO1 = 20,4;**

e que em outro lugar do programa você digitou errado

**CIRCUM = 6,28 \* RAIOI;**

- Você involuntariamente substituiu o numeral 1 pela letra I (I maiúsculo). Essa outra linguagem criaria uma nova variável chamada **RAIOI** e usaria qualquer valor que tivesse (talvez zero, talvez lixo - **garbage**). A variável **CIRCUM** receberia o valor errado e você pode ter muita dificuldade para descobrir o porquê. Isso não pode acontecer em C (a menos que você tenha sido bobo o suficiente para declarar dois nomes de variáveis semelhantes) porque o compilador irá reclamar quando o **RAIOI** não declarado aparecer.

- Seu programa C não será compilado se você não declarar suas variáveis. Se as razões anteriores não conseguirem convencê-lo, você deve pensar seriamente nisso.

Dado que você precisa declarar suas variáveis, para onde elas vão? Como mencionado anteriormente, C antes de C99 exigia que as declarações fossem no início de um bloco. Uma boa razão para seguir esta prática é que agrupar as declarações torna mais fácil ver o que o programa está fazendo. Claro, há também uma boa razão para espalhar as suas declarações, como a C99 agora permite. A ideia é declarar variáveis antes de você estar pronto para atribuir um valor a elas. Isso torna mais difícil esquecer de dar-lhes um valor. Na prática, muitos compiladores ainda não suportam a regra C99.

## Atribuição

```
num = 1;
```

A próxima linha do programa é uma instrução de atribuição, uma das operações básicas em C. Este exemplo específico significa “atribuir o valor **1** à variável **num**”. A instrução anteriormente “**int num;**” significa **reservar espaço na memória do computador** para a variável **num**, enquanto a próxima linha “**num = 1;**” seria a atribuição que armazena um valor nesse local.

Você pode atribuir a **num** um valor diferente posteriormente, se desejar; é por isso que **num** é denominado **variável**. Porque varia de valor! Observe que a instrução de atribuição atribui um valor do lado direito para o lado esquerdo. Além disso, a instrução é completada com **ponto-e-vírgula**, conforme mostrado na imagem a seguir.

num = 1;

Operador de Atribuição

O diagrama mostra a instrução de atribuição 'num = 1;' em uma fonte grande. Abaixo dela, uma seta cinza aponta para cima, direcionada especificamente para o caractere '='. Sobre a seta, o texto 'Operador de Atribuição' está escrito em uma fonte menor e em negrito.

*Figura 36. A instrução de atribuição é uma das operações básicas do C.*



## A função printf()

```
printf("Eu sou um simples ");  
printf("computador.\n");  
printf("Meu número favorito eh %d porque eh o primeiro.\n", num);
```

Todas essas linhas usam uma função C padrão chamada **printf()**. Os parênteses significam que **printf** é um nome de função. O conteúdo entre parênteses é a informação passada da função **main()** para a função **printf()**. Por exemplo, a primeira linha passa a frase *“Eu sou um simples”* para a função **printf()**. Essa **informação** é chamada de **argumento** ou, mais detalhadamente, de **argumento real** de uma função (veja a Figura a seguir). (C usa os termos argumento e parâmetro para distinguir entre um valor específico enviado para uma função e uma variável na função usada para armazenar o valor; O argumento é o que é enviado na chamada de uma função! O Parâmetro é quando existe uma variável da função que recebe as informações do argumento na chamada. O que a função **printf()** faz com este argumento? Ele analisa tudo o que está entre aspas duplas e imprime esse texto na tela.



*Figura 37. A função printf() com um argumento.*

Esta primeira linha **printf()** é um exemplo de como você chama ou invoca uma função em C. Você precisa digitar apenas o nome da função, colocando o(s) argumento(s) desejado(s) entre parênteses. Quando o programa atinge esta linha, o controle é transferido para a função nomeada (**printf()** neste caso). Quando a função termina o que quer que faça, o controle é retornado para a função original (na chamada) — **main()**, neste exemplo.

E a próxima linha **printf()**? Tem os caracteres `\n` incluídos entre aspas e eles não foram impressos! O que está acontecendo? O símbolo `\n` significa iniciar uma nova linha. A combinação `\n` (digitada como dois caracteres) representa um único caractere chamado caractere de nova linha. Para **printf()**, significa *“iniciar uma nova linha na margem esquerda”*. Em outras palavras, imprimir o caractere de nova linha executa a mesma função que pressionar a tecla **Enter** de um teclado típico. Por que não usar a tecla **Enter** ao digitar o argumento **printf()**? Isso seria interpretado como um comando imediato para o seu editor, não como uma instrução a ser armazenada no seu código-fonte. Em outras palavras, ao pressionar a tecla **Enter**, o editor sai da linha atual na qual você está

trabalhando e inicia uma nova. O caractere de nova linha, entretanto, afeta a forma como a saída do programa é exibida.

O caractere de nova linha é um exemplo de **sequência de escape**. Uma **sequência de escape** é usada para representar caracteres difíceis ou impossíveis de digitar. Outros exemplos são `\t` para **Tab** (Tabulação) e `\b` para **Backspace** (Voltar para trás). Em cada caso, a sequência de escape começa com o caractere de barra invertida, `\`. Depois voltaremos a esse assunto!

Bem, isso explica por que as três instruções `printf()` produziram apenas duas linhas: A primeira instrução de impressão não tinha um caractere de nova linha, mas a segunda e a terceira tinham.

A linha `printf()` final traz outra estranheza: o que aconteceu com `%d` quando a linha foi impressa? Como você deve se lembrar, a saída para esta linha foi:

**Meu numero favorito eh 1 porque eh o primeiro.**

Aha! O dígito **1** foi substituído pelo grupo de símbolos `%d` quando a linha foi impressa e **1** foi o valor da variável `num`. O `%d` é um local no argumento reservado para mostrar onde o valor de `num` deve ser impresso. Esta linha é semelhante à seguinte instrução **BASIC**:

```
PRINT "Meu numero favorito eh "; num; " porque eh o primeiro."
```

A versão C faz um pouco mais do que isso! Na verdade, o caractere `%` alerta o programa que uma variável deve ser impressa naquele local, e o `d` diz para ele imprimir a variável como um **número inteiro decimal (base 10)**. A função `printf()` permite diversas escolhas para o formato das variáveis impressas, incluindo **números inteiros hexadecimais (base 16)** e **números com casas decimais**. Na verdade, o `f` em `printf()` é um lembrete de que esta é uma função de impressão de **formatação**. Cada **tipo de dados** tem seu **próprio especificador** – à medida que as explicações introduzem novos tipos, eles também apresentam os especificadores apropriados.

## Declaração de retorno

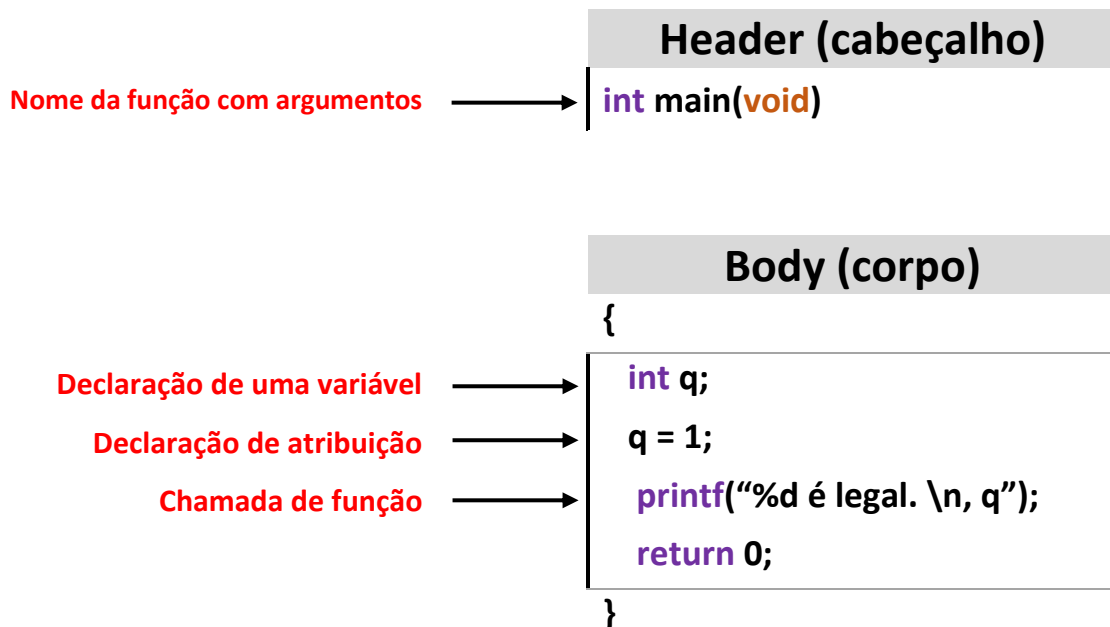
```
return 0;
```

Esta declaração de retorno é a declaração final do programa. O `int`, na linha `int main(void)`, significa que a função `main()` deve retornar um **número inteiro**. O padrão C exige que `main()` se comporte dessa maneira. As funções C que retornam valores fazem isso com uma instrução `return`, que consiste apenas na **palavra-chave return**, seguida pelo **valor retornado**, e depois seguido por um **ponto-e-vírgula**. Se você deixar de fora a instrução `return` para `main()`, o programa retornará **0** quando chegar ao fechamento `}`. Portanto, você pode omitir a instrução `return` no final de `main()`. No entanto, você não pode omiti-lo de outras funções, por isso é mais consistente usá-lo em `main()` também. Neste ponto, você pode considerar a instrução `return` em `main()` como algo necessário para consistência lógica, mas ela tem uso prático com alguns sistemas operacionais, incluindo **Linux** e **Unix**.

## Parou em: The Structure of a Simple Program

## A estrutura de um programa simples

Agora que viu um exemplo específico, você está pronto para algumas regras gerais sobre programas **C**. Um programa consiste em uma coleção de uma ou mais funções, uma das quais deve ser chamada **main()**. A descrição de uma função consiste em um **cabeçalho** e um **corpo**. O **cabeçalho da função** contém o **nome da função** junto com informações sobre o tipo de informação passada para a função e retornada pela função. Você pode reconhecer o nome de uma função pelos **parênteses**, que podem estar vazios. O **corpo** é colocado entre colchetes (**{}**) e consiste em uma série de instruções, cada uma terminada por ponto e vírgula (veja a **Figura 38**). O exemplo deste capítulo tinha uma declaração, anunciando o nome e o tipo da variável que está sendo usada. Em seguida, ele tinha uma instrução de atribuição dando um valor à variável. Em seguida, havia três instruções **print**, cada uma chamando a função **printf()**. As instruções **print** são exemplos de instruções de **chamada de função**. Finalmente, **main()** termina com uma instrução **return**. Resumindo, um programa **C** padrão simples deve usar o seguinte formato:



*Figura 38. Uma função possui um cabeçalho e um corpo.*

## Dicas para tornar seus programas legíveis

Tornar seus programas legíveis é uma boa prática de programação. Um programa legível é muito mais fácil de entender e facilita a correção ou modificação. O ato de tornar um programa legível também ajuda a esclarecer o seu próprio conceito do que o programa faz.

Você já viu duas técnicas para melhorar a legibilidade: Escolha nomes de variáveis significativos e use comentários. Observe que essas duas técnicas se complementam. Se você der a uma variável o nome `largura`, não precisa de um comentário dizendo que essa

variável representa uma largura, mas uma variável chamada `rotina_de_video_4` clama por uma explicação sobre o que a rotina de vídeo 4 faz.

Outra técnica envolve o uso de linhas em branco para separar uma seção conceitual de uma função de outra. Por exemplo, o programa de exemplo simples possui uma linha em branco separando a seção de declaração da seção de ação. C não requer a linha em branco, mas melhora a legibilidade.

Uma quarta técnica é usar uma linha por instrução. Novamente, esta é uma convenção de legibilidade, não um requisito C. O C tem um formato de formato livre. Você pode colocar várias declarações em uma linha ou espalhar uma declaração por várias. O código a seguir é legítimo, mas é muito feio:

```
#include<stdio.h>

int main(void) {int four; four
=
4
;
printf(
    "%d\n",
    four); return 0;}
```

Os pontos e vírgulas informam ao compilador onde uma instrução termina e a próxima começa, mas a lógica do programa fica muito mais clara se você seguir as convenções usadas no exemplo deste capítulo (veja o código a seguir, na Figura 39).

```
#include<stdio.h>

/* Converter 2 braçais em pés */ ----- Use comentários

int main(void)
{
    int feet, fathoms; ----- escolha nomes significativos
                           ----- use o espaço

    fathoms = 2;
    feet = 6 * fathoms; ----- uma instrução por linha
    printf("Existem %d pes em %d braçais!\n", feet, fathoms);
    return 0;
}
```

*Figura 39. Tornando seu programa legível.*

## Dando mais um passo no uso de C

O primeiro exemplo de programa foi bastante fácil, e o próximo exemplo, mostrado no exemplo a seguir, no código 2-1, não é muito mais difícil.

### *Código 2-1*

```
// fathm_ft.c ==> converte 2 braçais (fathoms) para pés (feet)

#include<stdio.h>

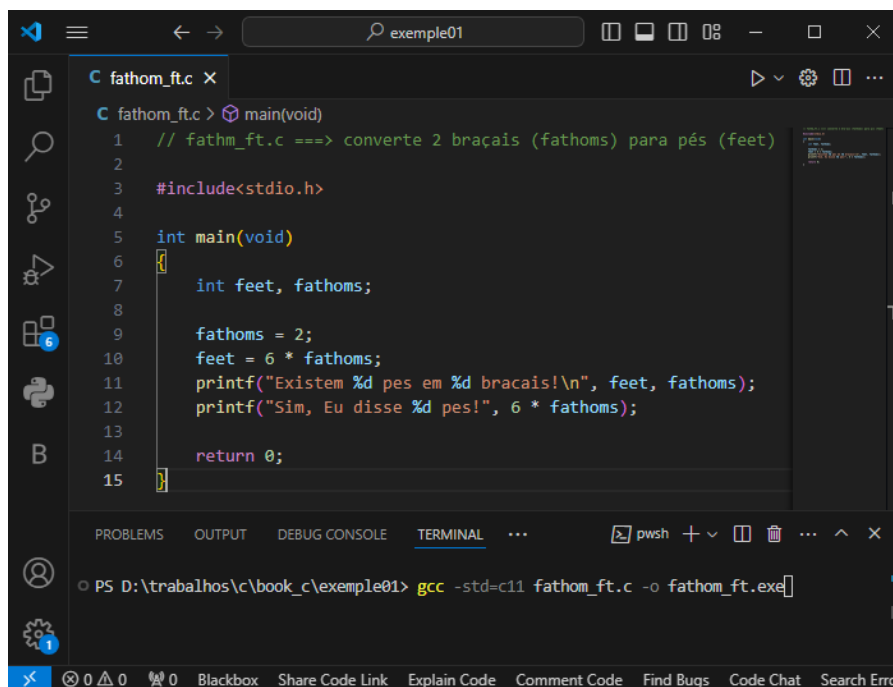
int main(void)
{
    int feet, fathoms;

    fathoms = 2;
    feet = 6 * fathoms;
    printf("Existem %d pes em %d braçais!\n", feet, fathoms);
    printf("Sim, Eu disse %d pes!", 6 * fathoms);

    return 0;
}
```

Compilando o programa pelo Terminal, digitando o comando (Veja a **Figura 40**):

***gcc -std=c11 fathom\_ft.c -o fathom\_ft.exe***



*Figura 40*

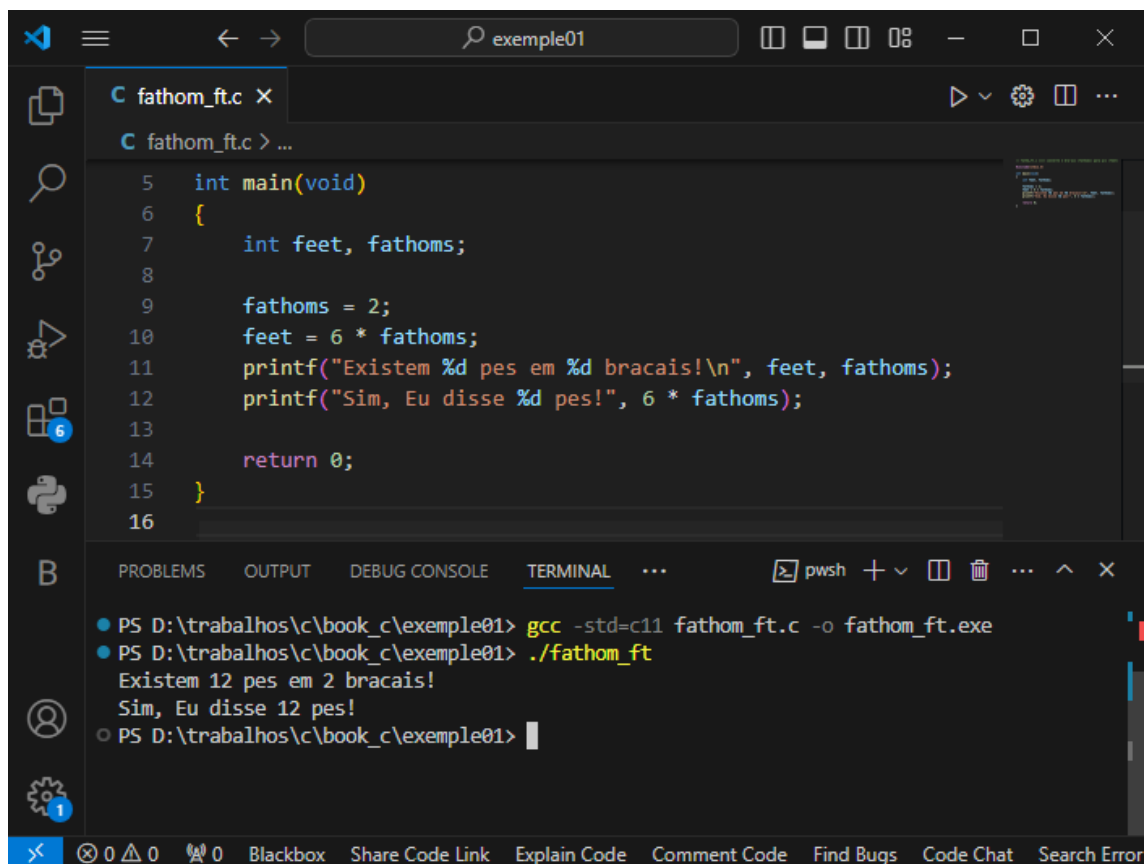
Se tudo ocorreu bem, digite o comando pelo terminal para rodar o programa (Veja a Figura 41):

***./fathom\_ft***

**RESULTADO NO TERMINAL:**

***Existem 12 pes em 2 bracaís!***

***Sim, Eu disse 12 pes!***



The screenshot shows the Visual Studio Code interface. The editor displays the source code for `fathom_ft.c`:

```
5 int main(void)
6 {
7     int feet, fathoms;
8
9     fathoms = 2;
10    feet = 6 * fathoms;
11    printf("Existem %d pes em %d bracaís!\n", feet, fathoms);
12    printf("Sim, Eu disse %d pes!", 6 * fathoms);
13
14    return 0;
15 }
16
```

The bottom panel shows the TERMINAL with the following commands and output:

```
PS D:\trabalhos\c\book_c\exemplo01> gcc -std=c11 fathom_ft.c -o fathom_ft.exe
PS D:\trabalhos\c\book_c\exemplo01> ./fathom_ft
Existem 12 pes em 2 bracaís!
Sim, Eu disse 12 pes!
PS D:\trabalhos\c\book_c\exemplo01>
```

*Figura 41.*

O que há de novo? O código fornece uma descrição do programa, declara múltiplas variáveis, faz algumas multiplicações e imprime os valores de duas variáveis. Vamos examinar esses pontos com mais detalhes.

## Documentação

Primeiro, o programa começa com um comentário (usando o novo estilo de comentário, com duas barras `//`) identificando o nome do arquivo e a finalidade do programa. Esse tipo de documentação de programa leva apenas um momento para ser feito e é útil posteriormente, quando você navega por vários arquivos ou os imprime.

## Múltiplas Declarações

A seguir, o programa declara duas variáveis em vez de apenas uma em uma única declaração. Para fazer isso, separe as duas variáveis (pés e braças) por uma vírgula na instrução de declaração. As declarações a seguir são equivalentes:

```
int feet, fathoms;  
Ou  
int fathoms;  
int feet;
```

## Multiplicação

Terceiro, o programa faz um cálculo. Ele aproveita o tremendo poder computacional de um sistema de computador para multiplicar 2 por 6. Em C, como em muitas linguagens, o asterisco `*` é o símbolo da multiplicação. Portanto, a declaração

```
feet = 6 * fathoms;
```

significa “procurar o valor da variável braças, multiplicá-lo por 6 e atribuir o resultado deste cálculo à variável pés”.

## Imprimindo vários valores

Finalmente, o programa faz um uso mais sofisticado de **`printf()`**. Se você compilar e executar o exemplo, a saída deverá ser semelhante a esta:

***Existem 12 pes em 2 braçais!***  
***Sim, Eu disse 12 pes!***

Desta vez, o código fez duas substituições na primeira utilização de **`printf()`**. O primeiro **`%d`** entre aspas foi substituído pelo valor da primeira variável (**`feet`**) na lista após o segmento citado, e o segundo **`%d`** foi substituído pelo valor da segunda variável (**`fathoms`**) na lista. Observe que a lista de variáveis a serem impressas vem no final da instrução, após a parte citada. Observe também que cada item é separado dos demais por uma vírgula.

O segundo uso de **`printf()`** ilustra que o valor impresso não precisa ser uma variável; só precisa ser algo, como **`6 * fathoms`**, que reduza a um valor do tipo certo.

Este programa tem escopo limitado, mas poderia formar o núcleo de um programa para converter braças em pés. Tudo o que é necessário é uma forma de atribuir valores

adicionais a variável “*feet*” de forma interativa; explicaremos como fazer isso em capítulos posteriores.

## Enquanto você faz isso – múltiplas funções

Até agora, esses programas usaram a função ***printf()*** padrão. O **Código 2-2** mostra como incorporar uma função própria — além de ***main()*** — em um programa.

*Código 2-2.*

```
// two_func.c -- um programa usando duas funções em um arquivo

#include <stdio.h>

/* ANSI/ISO C - Protótipo de função - Apenas a declaração */
void butler(void);

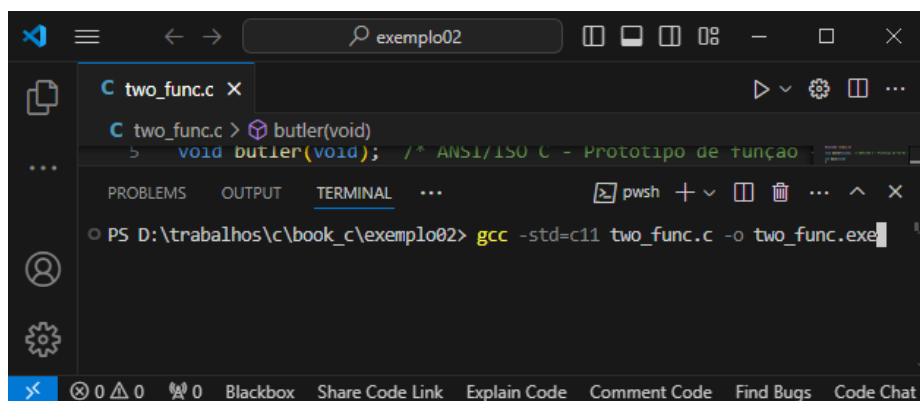
int main(void)
{
    printf("Vou convocar a funcao de mordomo.\n");
    butler();
    printf("Sim. Traga-me um pouco de cha e DVDs gravaveis.\n");

    return 0;
}

void butler (void) /* Aqui começa a definição da função*/
{
    printf("Chamou, senhor?\n");
}
```

Digite os comandos a seguir no terminal do Visual Studio Code para compilar o código anterior, e dê um “Enter” (depois veja a **Figura 42**):

***gcc -std=c11 two\_func.c -o two\_func.exe***



*Figura 42. Compilando o programa.*



Se tudo ocorreu bem, então digite na linha seguinte do Terminal, o seguinte comando:

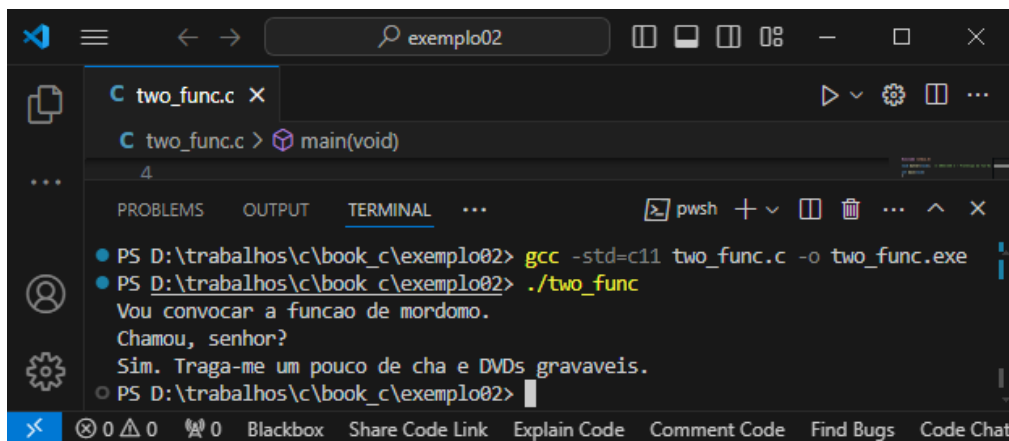
***./two\_func***

***Mensagem de saída no Terminal:***

***Vou convocar a funcao de mordomo.***

***Chamou, senhor?***

***Sim. Traga-me um pouco de cha e DVDs gravaveis.***



```
PS D:\trabalhos\c\book_c\exemplo02> gcc -std=c11 two_func.c -o two_func.exe
PS D:\trabalhos\c\book_c\exemplo02> ./two_func
Vou convocar a funcao de mordomo.
Chamou, senhor?
Sim. Traga-me um pouco de cha e DVDs gravaveis.
PS D:\trabalhos\c\book_c\exemplo02>
```

*Figura 43. Mensagem na Tela do Terminal.*

A função ***butler()*** aparece **três vezes** neste programa. A primeira aparição está no protótipo, bem no início, depois da diretiva de pré-processador (***#include<>***) e antes da função principal ***main()***, que informa ao compilador sobre as funções a serem utilizadas. A segunda aparição está dentro do corpo de ***main()***, entre as chaves “{ }”, na forma de uma chamada de função. Por fim, o programa apresenta a definição da função ***butler()***, depois do corpo da função ***main()***, que é o código fonte da própria função. Vejamos cada uma dessas três aparições.

O padrão **C90** adicionou protótipos, porém compiladores mais antigos podem não os reconhecer. (**Diremos a você o que fazer ao usar esses compiladores em breve.**)

Um **protótipo de função** declara ao compilador que você está usando uma função específica, por isso é chamado de **declaração de função**. Ele também especifica propriedades da função.

Por exemplo, o primeiro **void** no protótipo da função ***butler()*** indica que ***butler()*** não possui um valor de retorno. (**Em geral, uma função pode retornar um valor para a função chamadora para seu uso, mas *butler()* não.**)

O segundo **void** — aquele em ***butler(void)*** — significa que a função ***butler()*** não tem argumentos. Portanto, quando o compilador atinge o ponto em ***main()*** onde ***butler()*** é usado, ele pode verificar se ***butler()*** é usado corretamente. Observe que **void** é usado para significar “**vazio**”, não “**inválido**”.

O C mais antigo suportava uma forma mais limitada de declaração de função na qual você apenas especificava o tipo de retorno, mas omitia a descrição dos argumentos:

```
void butler();
```

O código C mais antigo usa declarações de função como a anterior em vez de protótipos de função. Os padrões C90, C99 e C11 reconhecem esta forma mais antiga, mas indicam que ela será eliminada com o tempo, portanto não a utilize. Se você herdar algum código C legado, você pode querer converter as declarações de estilo antigo em protótipos. Os capítulos posteriores deste livro voltam à prototipagem, às declarações de funções e aos valores de retorno.

Em seguida, você invoca **butler()** em **main()** simplesmente fornecendo seu nome, incluindo parênteses. Quando **butler()** termina seu trabalho, o programa passa para a próxima instrução em **main()**.

Finalmente, a função **butler()** é definida da mesma maneira que **main()**, com um **cabeçalho de função** e o **corpo entre colchetes**. O cabeçalho repete as informações fornecidas no protótipo: **butler()** não aceita argumentos e não possui valor de retorno. Para compiladores mais antigos, omite o segundo **void**.

Um ponto a ser observado é que é o local da chamada **butler()** em **main()** — e não o local da definição **butler()**, que fica abaixo do corpo de **main()**, no arquivo — que determina quando a **função butler()** é executada. Você poderia, por exemplo, colocar a definição **butler()** acima da definição **main()** neste programa, e o programa ainda funcionaria da mesma forma, com a função **butler()** executada entre as duas chamadas para **printf()** em **main()**. Lembre-se de que todos os programas C começam a execução com **main()**, não importa onde **main()** esteja localizado nos arquivos do programa. No entanto, a prática usual em C é listar **main()** primeiro porque normalmente fornece a estrutura básica para um programa.

O padrão C recomenda que você forneça **protótipos de função** para todas as funções que você usa. Os arquivos de inclusão padrão cuidam dessa tarefa para as **funções da biblioteca padrão**. Por exemplo, no padrão C, o arquivo **stdio.h** possui um protótipo de função para **printf()**. O exemplo final no Capítulo 6 mostrará como estender a prototipagem para funções não nulas, e o Capítulo 9 cobre funções completamente.

## RESUMINDO:

Os protótipos de função em C são declarações de funções que especificam seu nome, tipo de retorno e os tipos de seus parâmetros, mas não necessariamente o corpo da função. Eles são geralmente colocados no início do arquivo fonte ou em arquivos de cabeçalho. A seguir estão as principais importâncias e vantagens dos protótipos de função:

### 1. Verificação de Tipos

Os protótipos de função permitem ao compilador realizar a verificação de tipos em tempo de compilação. Se houver uma chamada de função com argumentos que não

correspondem aos tipos especificados no protótipo, o compilador gerará um erro, prevenindo bugs que poderiam ocorrer em tempo de execução.

## 2. Declarações Antecipadas

Ao usar protótipos de função, você pode declarar funções em qualquer ordem no arquivo de origem. O compilador saberá sobre a existência de uma função antes de sua definição real, permitindo chamadas de funções que ainda não foram definidas no ponto em que são chamadas.

## 3. Modularidade e Organização

Os protótipos de função ajudam a modularizar o código. Funções podem ser definidas em arquivos de origem separados e seus protótipos podem ser incluídos em arquivos de cabeçalho. Isso facilita a manutenção e organização do código, além de promover a reutilização de funções em diferentes partes do programa ou em programas diferentes.

## 4. Documentação

Protótipos de função servem como uma forma de documentação para outros desenvolvedores, indicando claramente quais funções estão disponíveis, seus tipos de retorno e os tipos dos parâmetros esperados. Isso é especialmente útil em projetos grandes e colaborativos.

## 5. Facilita a Detecção de Erros

Ao especificar protótipos de função, erros relacionados a argumentos incorretos ou número errado de argumentos são detectados em tempo de compilação, reduzindo o número de bugs difíceis de identificar que poderiam ocorrer em tempo de execução.

```
#include <stdio.h>

// Protótipo de função
int soma(int a, int b);

int main() {
    int resultado = soma(5, 3);
    printf("Resultado: %d\n", resultado);
    return 0;
}

// Definição da função
int soma(int a, int b) {
    return a + b;
}
```

Neste exemplo, o protótipo `int soma(int a, int b);` declara a função `soma` antes de sua definição, permitindo que ela seja chamada no `main` antes da definição real da função. Se houvesse um erro nos tipos dos argumentos ou no número de argumentos na chamada da função `soma`, o compilador emitiria um erro.

Em resumo, os protótipos de função são uma prática recomendada em C que ajuda a garantir a integridade e a clareza do código, além de proporcionar benefícios de verificação de tipos e organização.

## EXEMPLO DE MODULARIZAÇÃO

Para modularizar um programa em C, você pode seguir essa estrutura dividindo seu código em vários arquivos, o que torna o código mais organizado e fácil de manter. Aqui está um exemplo de como você pode fazer isso:

### Passo 1: Arquivo de Cabeçalho (`soma.h`)

Este arquivo conterá o protótipo da função `soma`. Veja que é somente o cabeçalho da função, ou seja, a sua assinatura. Veja, também, no código abaixo a condição (`#ifndef ... #endif`) é uma diretiva usada constantemente para **impedir** que arquivos de cabeçalho sejam incluídos várias vezes no mesmo arquivo-fonte.

```
// soma.h
// Essa diretiva abaixo impede várias inclusões deste arquivo
#ifndef SOMA_H // Se não definida SOMA_H
#define SOMA_H // Então defina SOMA_H

int soma(int a, int b);

#endif // Fim do ifndef
```

### Passo 2: Arquivo de Implementação da Função (`soma.c`)

Este arquivo conterá a definição ou implementação (o que essa função deve fazer) da função `soma`. Não esqueça de acrescentar na diretiva `#include` o nome do arquivo de cabeçalho entre as aspas duplas. Não coloque o nome do arquivo de cabeçalho que você criou entre os símbolos de **menor** e **maior** (`<>`). Os arquivos entre esses sinais são reservados aos arquivos da pasta **include**, que possuem funções padrões, voltadas para a linguagem C.

```
// soma.c
#include "soma.h"

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

## Passo 3: Arquivo Principal (`main.c`)

Este arquivo conterá a função `main()` e incluirá o cabeçalho com o protótipo da função. Novamente, veja que a diretiva **#include** possui o arquivo de cabeçalho padrão entre **menor** e **maior** (`<stdio.h>`) e o arquivo que você criou dentro da sua pasta de projeto ("`soma.h`")

```
// main.c
#include <stdio.h>
#include "soma.h"

int main() {
    int resultado = soma(5, 3);
    printf("Resultado: %d\n", resultado);
    return 0;
}
```

## Compilação

Para compilar esses arquivos, você precisa passar todos eles para o compilador. Se você estiver usando `gcc`, o comando será:

*`gcc main.c soma.c -o programa`*

**OBS:** Não se compila arquivos de cabeçalho. Neste caso, apenas os arquivos com extensão `“.c”`

## Explicação:

1. **Arquivo de Cabeçalho (`soma.h`):**
  - Contém o protótipo da função `soma`.
  - Utiliza guardas de inclusão (`#ifndef`, `#define`, `#endif`) para evitar múltiplas inclusões do mesmo arquivo.
2. **Arquivo de Implementação da Função (`soma.c`):**
  - Inclui o arquivo de cabeçalho `soma.h` para garantir que a definição da função `soma` corresponda ao protótipo.
  - Define a função `soma`.
3. **Arquivo Principal (`main.c`):**
  - Inclui `stdio.h` para utilizar `printf`.
  - Inclui `soma.h` para utilizar o protótipo da função `soma`.
  - Define a função `main` que chama a função `soma`.

## Vantagens:

- **Modularidade:** Separar o código em diferentes arquivos torna o programa mais fácil de entender, manter e modificar.
- **Reutilização:** Funções como `soma` podem ser reutilizadas em outros programas simplesmente incluindo `soma.h` e `soma.c`.
- **Compilação Incremental:** Apenas os arquivos modificados precisam ser recompilados, o que pode economizar tempo durante o desenvolvimento.

Seguindo esta abordagem, você garante um código bem organizado e modular, facilitando a colaboração em projetos maiores e a manutenção a longo prazo.

## EXEMPLO DE UM PROJETO USANDO UMA ESTRUTURA DE DIRETÓRIOS

Vamos organizar um projeto simples de C com uma estrutura de diretórios clara e explicações sobre como usar o GCC para compilar e criar bibliotecas. A estrutura de diretórios vai ajudar a manter o projeto organizado, especialmente à medida que ele cresce.

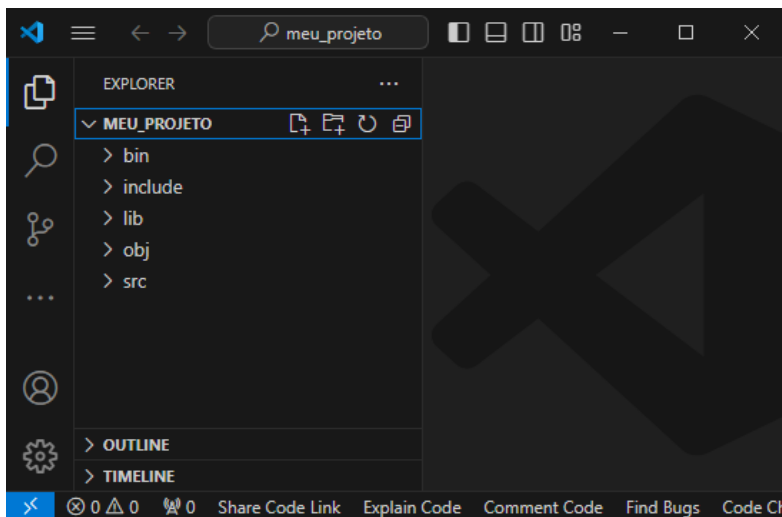
Na pasta **meu\_projeto**, temos as pastas **include** (*headers* - para arquivos de cabeçalho), **src** (*source* - arquivos fontes), **obj** (*objects* - arquivos objeto), **lib** (*.dll* - bibliotecas dinâmicas) e **bin** (arquivos executáveis)

## Estrutura de Diretórios

Aqui está uma estrutura de diretórios sugerida para o projeto:

```
meu_projeto/
├── include/    # Headers (.h)
│   └── soma.h
├── src/       # Arquivos fonte (.c)
│   ├── soma.c
│   └── main.c
├── obj/       # Arquivos objeto (.o)
├── lib/       # Bibliotecas dinâmicas (.dll)
└── bin/       # Executáveis
```

Veja a estrutura de diretórios no **Visual Studio Code**, na imagem a seguir (Figura 44):



*Figura 44. Estrutura de diretórios.*

## Passo a Passo para Compilar e Linkar Usando GCC

### 1. Criação dos Arquivos de Código-Fonte e Cabeçalho

Crie os seguintes arquivos com o conteúdo especificado:

**Arquivo** `include/soma.h`:

```
#ifndef SOMA_H
#define SOMA_H

#ifdef _WIN32
    #ifdef SOMA_EXPORTS
        #define SOMA_API __declspec(dllexport)
    #else
        #define SOMA_API __declspec(dllimport)
    #endif
#else
    #define SOMA_API
#endif

SOMA_API int soma(int a, int b);

#endif
```

Veja a imagem a seguir (**Figura 45**):

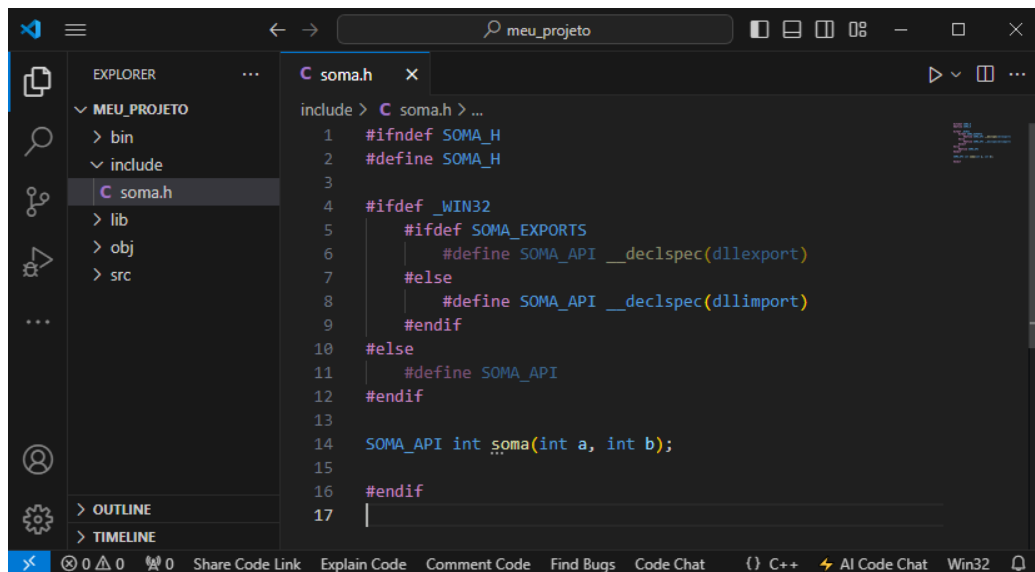


Figura 45.

Esse arquivo é um exemplo de cabeçalho em C/C++ que define uma função de soma (`soma`) e usa diretivas de pré-processamento para gerenciar a exportação/importação dessa função em diferentes sistemas operacionais.

Vamos analisar o arquivo linha por linha:

1. `#ifndef SOMA_H`
  - Isso verifica se o arquivo de cabeçalho `SOMA_H` ainda não foi incluído anteriormente no código. Se ainda não foi incluído, o conteúdo dentro deste bloco será processado.
2. `#define SOMA_H`
  - Isso define o identificador `SOMA_H`, indicando que o arquivo de cabeçalho foi incluído e seu conteúdo deve ser processado.
3. `#ifdef _WIN32`
  - Isso verifica se o código está sendo compilado em um ambiente Windows.
4. `#ifdef SOMA_EXPORTS`
  - Isso verifica se a macro `SOMA_EXPORTS` está definida. Essa macro geralmente é definida ao compilar um arquivo de código-fonte que exporta a função `soma`.
5. `#define SOMA_API __declspec(dllexport)`
  - Se `SOMA_EXPORTS` estiver definida, isso define a macro `SOMA_API` como `__declspec(dllexport)`. Isso indica que a função `soma` será exportada de um DLL no Windows.
6. `#else`
  - Se `SOMA_EXPORTS` não estiver definida, este bloco será executado.
7. `#define SOMA_API __declspec(dllimport)`
  - Isso define a macro `SOMA_API` como `__declspec(dllimport)`, indicando que a função `soma` será importada de um DLL no Windows.
8. `#endif`
  - Fecha o bloco de verificação `#ifdef _WIN32`.
9. `#else`



- 
- Se o sistema operacional não for Windows, este bloco será executado.
10. `#define SOMA_API`
- Isso define `SOMA_API` como vazio, indicando que não há necessidade de exportar/importar a função `soma` em outros sistemas operacionais.
11. `#endif`
- Fecha o bloco `#ifndef SOMA_H`.
12. `SOMA_API int soma(int a, int b);`
- Esta linha declara a função `soma` com a assinatura `int soma(int a, int b);`, usando `SOMA_API` para definir se a função será exportada, importada ou não em diferentes sistemas operacionais.

Em resumo, esse arquivo de cabeçalho é usado para definir a função `soma` de forma que ela possa ser exportada ou importada corretamente dependendo do sistema operacional em que o código está sendo compilado. Isso é útil ao trabalhar com bibliotecas compartilhadas (DLLs) no Windows, por exemplo.

---

Arquivo `src/soma.c`:

```
#include "soma.h"

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

Olhe a imagem abaixo (Figura 46):

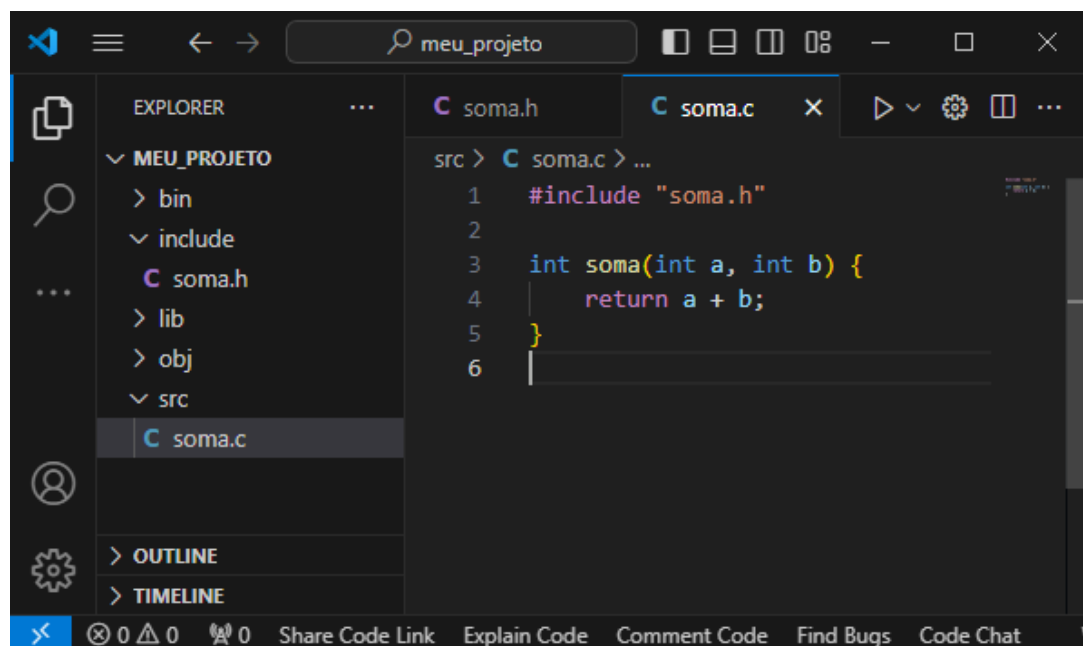


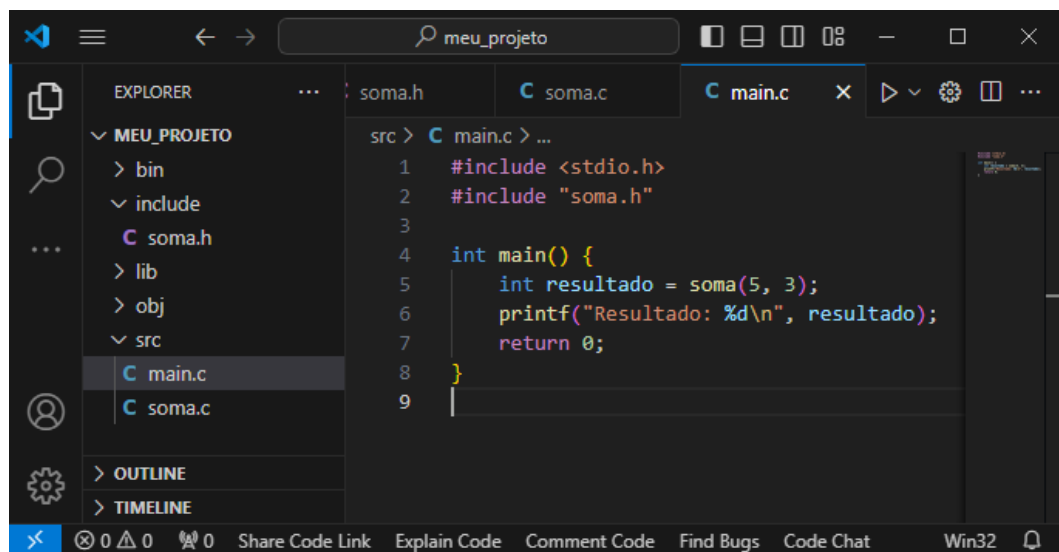
Figura 46.

Arquivo `src/main.c`:

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

int main() {
    int resultado = soma(5, 3);
    printf("Resultado: %d\n", resultado);
    return 0;
}
```

A imagem a seguir (**Figura 47**):



*Figura 47.*

## 2. Compilação do Arquivo Fonte para Objeto

Compile o arquivo `soma.c` para gerar o arquivo objeto `soma.o`. O arquivo objeto será colocado na pasta `obj/`.

Ao compilar o arquivo `soma.c`, precisamos definir o símbolo `SOMA_EXPORTS` para indicar que estamos criando a DLL.

```
gcc -c -fPIC src/soma.c -o obj/soma.o -Iinclude -DSOMA_EXPORTS
```

## 3. Criação da Biblioteca Dinâmica (DLL)

Crie a biblioteca compartilhada `soma.dll` usando o arquivo objeto `soma.o`.

```
gcc -shared -o lib/soma.dll obj/soma.o
```

#### 4. Compilação e Linkagem do Programa Principal

Compile e linke o programa principal `main.c`, especificando o diretório onde está a **DLL** (`lib/`) e os arquivos de cabeçalho (`include/`). O executável será colocado na pasta `bin/`.

#### 5. Tentar EXECUTAR o programa com o comando a seguir.

```
./bin/programa.exe
```

Veja que nada acontece! Executamos o aplicativo através da pasta da raiz `MEU_PROJETO`.

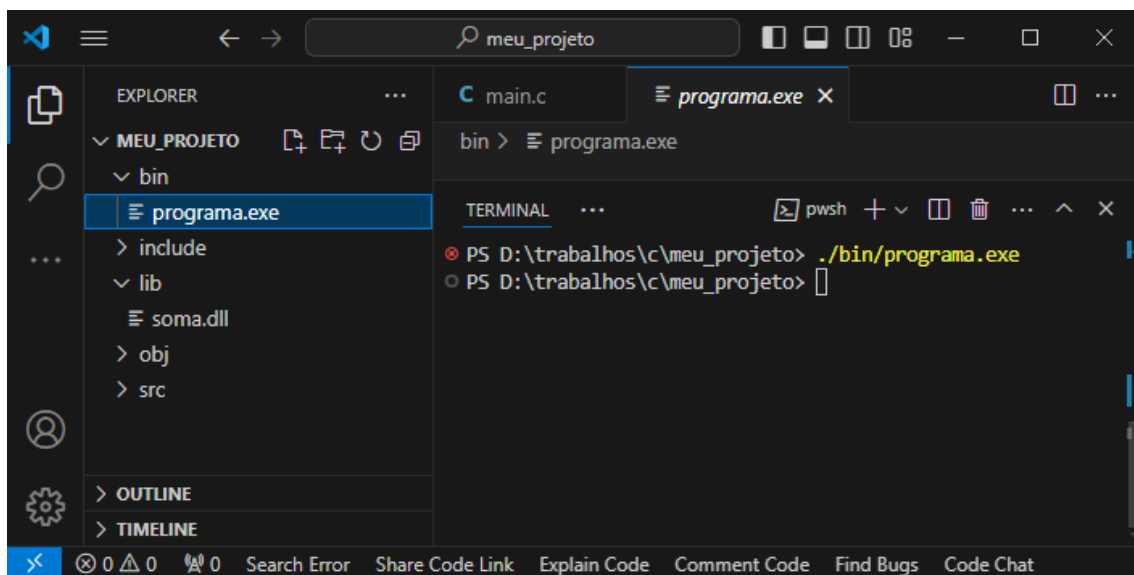
`./` → Comando para executar

`/bin` → Diretório bin

`programa.exe` → Arquivo Executável

Caminho completo para executar o arquivo: `./bin/programa.exe`

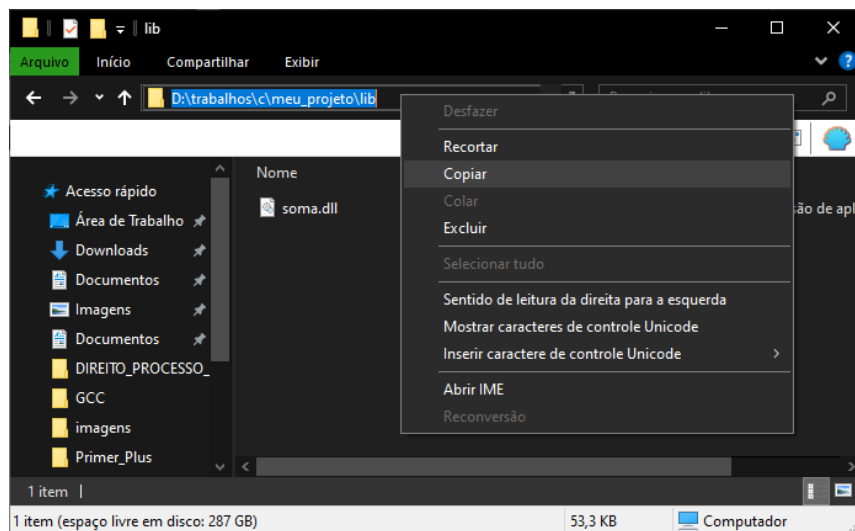
Veja a Figura 48 abaixo.



*Figura 48. Ao executar a aplicação, NADA ACONTECE.*

Se nada aconteceu, ou seja, não apareceu nada no Terminal, então o que devemos fazer? Sabemos que a função “*soma()*” está contida agora num arquivo de biblioteca dinâmica que não fica contida no executável. Quando este é chamado a carregar na memória, ele tenta procurar a função “*soma()*”, mas ONDE? Ele tenta procurar o arquivo no sistema, mas não o encontra. Na verdade, ele precisa encontrar um caminho ou PATH.

Precisamos entrar na pasta MEU\_PROJETO, e dentro da pasta lib, usando o explorador de arquivos do Windows 10 ou superior, e copiar seu caminho. No meu caso, o caminho é esse: **D:\trabalhos\c\meu\_projeto\lib**



*Figura 49. Copiar o caminho da pasta lib que está dentro de MEU\_PROJETO.*

Para executar o **programa.exe** e incluir a pasta **lib** no caminho de busca das bibliotecas dinâmicas, você pode usar a variável de ambiente **PATH** ou **LD\_LIBRARY\_PATH** no **PowerShell**. Aqui está um comando que você pode tentar:

```
$Env:PATH += ";D:\trabalhos\c\meu_projeto\lib"; ./bin/programa.exe
```

Substitua **D:\trabalhos\c\** pelo caminho completo até o diretório **MEU\_PROJETO**. Isso adicionará temporariamente o diretório **lib** ao **PATH**, permitindo que o sistema encontre a **soma.dll** quando você executar o **programa.exe**. Veja que o **PATH**, dentro de aspas duplas, começa com ";". A operação **+=** vai concatenar temporariamente o caminho especificado no **PATH**.

**Exemplo:** `$Env:PATH += ";pasta01\pasta02\pasta03\...\\"; ./pasta04/arquivo.exe`

Lembre-se de que essa alteração no **PATH** é temporária e só dura enquanto a sessão do **PowerShell** estiver aberta. Se você fechar e abrir uma nova sessão, precisará executar o comando novamente.

Executando...

```
$Env:PATH += ";D:\trabalhos\c\meu_projeto\lib"; ./bin/programa.exe
```

**Resultado no Terminal:**

```
Programa iniciado.  
Resultado da soma: 8  
Programa finalizado.
```

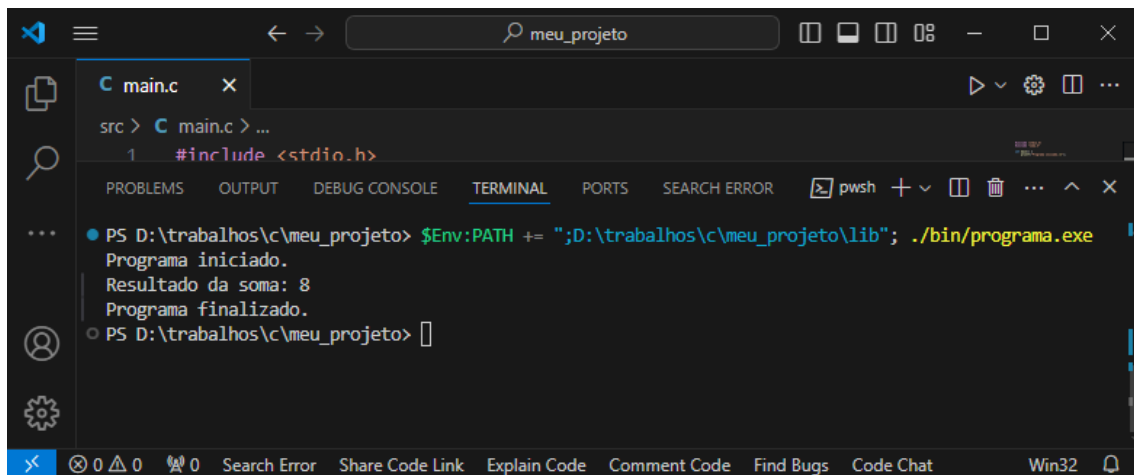


Figura 50. Incluindo um PATH para executar a aplicação.

Podemos também, no lugar de incluir um **PATH**, copiar e colar o arquivo **soma.dll**, que está na pasta **lib**, para a pasta **bin**.

#### 6. Copiar o arquivo “soma.dll” para a pasta a bin.

Isso é necessário porque a função soma está contida num arquivo separado (soma.dll) e é externa ao executável. O arquivo executável (programa.exe) somente inicia a aplicação, mas precisa encontrar a função soma que está dentro desse arquivo dinâmico com extensão “.dll”. Faça manualmente ou utilize o comando **copiar (cp)** o arquivo soma.dll da pasta origem (**lib/soma.dll**) para a pasta destino (**bin/**). (Veja a Figura 51):

```
cp lib/soma.dll bin/
```

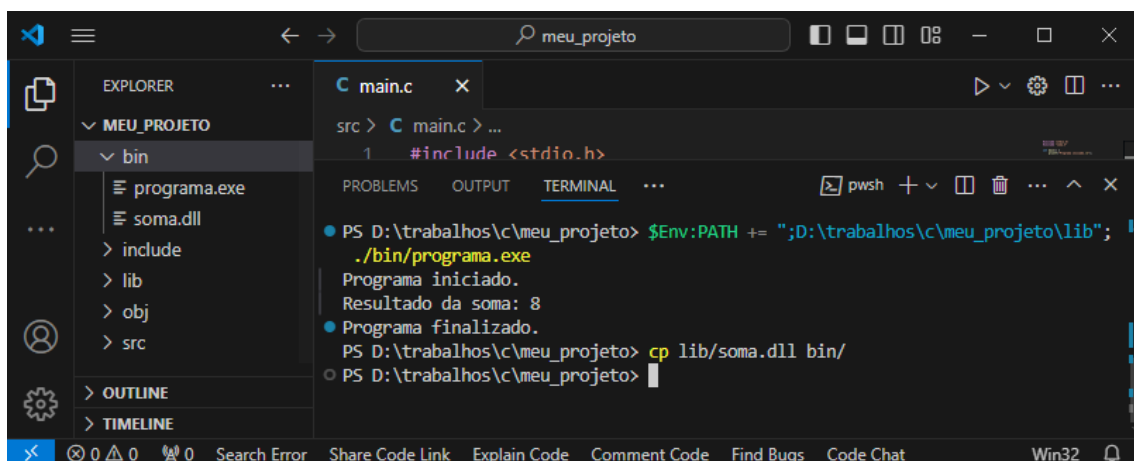
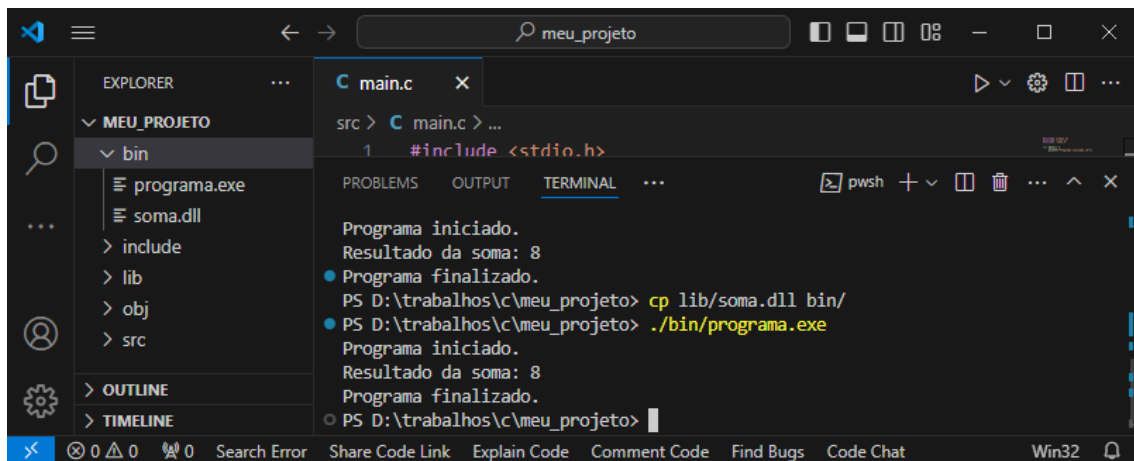


Figura 51. Comando cp para copiar da origem para o destino.

Certifique-se de que **soma.dll** está no mesmo diretório que **programa.exe** ou no **PATH** do sistema. Então, execute o programa (Veja a Figura 52):

```
./bin/programa.exe
```



*Figura 52. Executando a aplicação.*

Como estamos usando o Windows 10 ou superior, poderíamos, no [Visual Studio Code](#), criar um arquivo de **script**, no caso o formato **JSON (JavaScript Object Notation)**, uma alternativa ao **XML**, para compilar todo o projeto e executá-lo apenas usando o botão “run”. Depois iremos entender como o arquivo **JSON** funciona no [Visual Studio Code](#). Mas a explicação é entender como as coisas funcionam, por isso é necessário quebrar um pouco a cabeça!

## Explicação

- **gcc -c -fPIC src/soma.c -o obj/soma.o -Iinclude -DSOMA\_EXPORTS:** A flag **-DSOMA\_EXPORTS** define o símbolo **SOMA\_EXPORTS** durante a compilação de **soma.c**, o que permite que **\_\_declspec(dllexport)** seja usado corretamente. Isso informa ao compilador que estamos exportando a função **soma** ao criar a DLL.
- **gcc -shared -o lib/soma.dll obj/soma.o:** Cria uma biblioteca compartilhada **soma.dll** a partir do arquivo objeto **soma.o**.
- **gcc -o bin/programa src/main.c -Iinclude -Llib -lsoma:** Compila e linka o programa principal **main.c**, incluindo headers (**-Iinclude**) e especificando a biblioteca (**-Llib -lsoma**).

## Mais sobre as flags do GCC

Aqui está uma visão geral das principais flags de linkagem mencionadas no documento:

- **-l:** Linka com uma biblioteca. Por exemplo, **-lsoma** linkaria com a biblioteca **soma.dll**.
- **-L:** Adiciona um diretório à lista de diretórios de pesquisa para bibliotecas.
- **-static:** Força a linkagem com bibliotecas estáticas.
- **-shared:** Cria uma biblioteca compartilhada (DLL no Windows). Informa ao GCC que queremos criar uma biblioteca compartilhada (DLL).
- **-o soma.dll:** Especifica o nome do arquivo de saída como **soma.dll**.
- **-fPIC (Position Independent Code)**, que é importante para criar bibliotecas compartilhadas.

- `-D` no `gcc` é usada para definir macros no código fonte durante a compilação.
- `-DSOMA_EXPORTS`: Define a macro `SOMA_EXPORTS` para controlar a exportação de funções.

## Sobre a flag `-fPIC`

Em ambiente Unix-Like (como Linux e MacOS), a flag `-fPIC` (Position Independent Code) no GCC (GNU Compiler Collection) é usada para gerar código que é independente da posição em que será carregado na memória. Isso é especialmente útil ao compilar bibliotecas compartilhadas (shared libraries).

### 1. Bibliotecas Compartilhadas:

- Quando você está criando uma biblioteca compartilhada (`.so`), é importante que o código dentro da biblioteca possa ser carregado em qualquer endereço na memória. Isso permite que a mesma biblioteca seja mapeada em diferentes processos sem precisar de modificações.

### 2. Segurança e Eficiência:

- Código independente da posição melhora a segurança, pois facilita a implementação de técnicas como **Address Space Layout Randomization** (ASLR), que dificulta ataques que dependem de endereços de memória conhecidos.
- Além disso, ajuda na eficiência ao reduzir o número de relocações necessárias quando o código é carregado, o que pode resultar em tempos de carregamento mais rápidos.

## Como a flag `-fPIC` Funciona no Linux

A flag `-fPIC` (Position Independent Code) é mais frequentemente utilizada em sistemas Unix-like, como Linux, para a criação de bibliotecas compartilhadas (shared libraries). Essa flag instrui o compilador a gerar código que não depende de uma localização fixa na memória, permitindo que a biblioteca compartilhada possa ser carregada em qualquer endereço de memória. Isso é crucial para bibliotecas compartilhadas, pois facilita o compartilhamento de uma única cópia da biblioteca entre múltiplos processos.

Normalmente, quando o código é compilado, ele contém endereços absolutos. Isso significa que, se o código for carregado em um endereço diferente daquele previsto na compilação, ele precisa ser ajustado (relocado). No entanto, com o código independente da posição:

- O código usa endereços relativos em vez de absolutos.
- Isso é conseguido através de tabelas de saltos e indireção, que permitem que o código funcione corretamente independentemente de onde ele é carregado na memória.

## Quando Usar `-fPIC` No Linux

- **Bibliotecas Compartilhadas (Shared Libraries):** Quando você está criando bibliotecas que serão carregadas dinamicamente durante a execução do programa (como `.so` no Linux e `.dll` no Windows), o uso de `-fPIC` garante que o código pode ser relocado.
- **Modularidade e Reutilização de Código:** Facilita a criação de componentes de software que podem ser facilmente compartilhados entre diferentes programas e processos.

```
# Compilar com -fPIC
gcc -c -fPIC mylib.c -o mylib.o
# Criar a biblioteca compartilhada
gcc -shared -o libmylib.so mylib.o
```

## Uso do `-fPIC` No Windows

No Windows, a flag `-fPIC` não é necessária para criar bibliotecas dinâmicas (`.dll`). A criação de uma DLL no Windows não requer explicitamente a geração de código independente de posição, como é necessário no Linux. Portanto, você pode simplificar os comandos de compilação **removendo** a flag `-fPIC`.

Por que então utilizamos no projeto anterior essa flag se não estamos num ambiente Unix-Like, como Linux? Foi um erro?

Não. Apenas utilizamos boa prática de programação na criação de bibliotecas dinâmicas que são essenciais para dispositivos móveis, também, em ambiente Unix-Like, futuramente.

## Conclusão

A flag `-fPIC` é essencial ao criar bibliotecas compartilhadas, pois permite que o código seja carregado em qualquer endereço na memória, melhorando a segurança e a eficiência do carregamento. Ela é usada para instruir o compilador a gerar código que utilize endereços **relativos** em vez de **absolutos**, facilitando a relocação do código durante a execução.

Se você está usando o `gcc` do MinGW-w64 no Windows, o processo de criação de bibliotecas compartilhadas (`.dll`) é similar ao de criar bibliotecas compartilhadas (`.so`) no Linux, mas com algumas diferenças específicas para o ambiente Windows, como já explicado anteriormente.

Portanto, no ambiente Windows, usando MinGW-w64 e `gcc`, você pode criar e usar DLLs de maneira semelhante ao processo em sistemas Unix-like, mas, novamente, com algumas



diferenças nas convenções e práticas específicas para Windows. A flag `-fPIC` ainda é usada para gerar código independentemente da posição, embora não seja estritamente necessária para Windows, não ter efeito algum, é uma boa prática.

## Exemplo no Windows (sem `-fPIC`)

1º) Comando para criar o arquivo **soma.o**:

```
gcc -c src/soma.c -o obj/soma.o -Iinclude -DSOMA_EXPORTS
```

2º) Criar o arquivo **soma.dll**:

```
gcc -shared -o lib/soma.dll obj/soma.o
```

3º) Criar o arquivo **main.o**:

```
gcc -c src/main.c -o obj/main.o -Iinclude
```

4º) Criar o arquivo executável programa.exe (veja que não é necessário colocar a extensão “.exe”):

```
gcc -o bin/programa obj/main.o -Iinclude -Llib -lsoma
```

5º) Copiar o arquivo soma.dll da pasta lib para a pasta bin (cp origem destino):

```
cp lib/soma.dll bin/
```

6º) Executar o programa:

```
./bin/programa
```

7º) Veja o Resultado no Terminal ou PowerShell:

```
Programa iniciado.  
Resultado da soma: 8  
Programa finalizado.
```

### **Flags Principais:**

- D → Defina a macro a ser utilizada;
- c → Compila o arquivo com extensão “.c” ou “.o”;
- o → Saída do arquivo (no caso, seria a saída programa.exe na pasta bin);
- I → Incluir a pasta “include” para encontrar o arquivo de cabeçalho;
- L → Adiciona o diretório lib. Essa flag é exclusiva do Linker;
- l → Linka com a biblioteca soma.dll (omitindo o prefixo `lib` e a extensão).

### **Finalizando**

Com essa estrutura e esses passos, você pode facilmente organizar e compilar seu projeto C usando o GCC. Se precisar adicionar mais funcionalidades ou bibliotecas, basta seguir essa mesma lógica, mantendo seu projeto bem estruturado e organizado.

## Recapitulando sobre comandos básicos do Terminal

### Comandos básicos de compilação

O compilador usado neste livro é o **gcc** (**GNU Compiler Collection**) [2]. Em seu modo mais simples basta relacionar os arquivos-fontes que se quer compilar. A opção de compilação `-o <arqsai>` faz o código executável ser armazenado no arquivo de nome `<arqsai>`. Sem ela o código executável é armazenado em um arquivo de nome `a.out`.

<b>gcc prog.c</b>	Compila o programa que está no arquivo <code>prog.c</code> e gera um executável, armazenando-o no arquivo <code>a.out</code> .
<b>gcc prog.c aux.c ent_sai.c</b>	Compila o programa cujo código está distribuído nos arquivos <code>prog.c</code> , <code>aux.c</code> e <code>ent_sai.c</code> e gera um executável no arquivo <code>a.out</code> .
<b>gcc -o prog prog.c</b>	Compila o programa que está no arquivo <code>prog.c</code> e gera um executável no arquivo <code>prog</code> .
<b>gcc prog.c aux.c ent_sai.c -o prg_exem</b>	Compila o programa cujo código está distribuído nos arquivos <code>prog.c</code> , <code>aux.c</code> e <code>ent_sai.c</code> e gera um executável no arquivo <code>prg_exem</code> .

O processo de compilação é aplicado a todos os arquivos indicados na linha de comando, preservando-se as etapas já realizadas para cada arquivo:

<b>gcc prog.s</b>	Compila o programa assembler que está no arquivo <b>prog.s</b> , executando as etapas de montagem e ligação. Gera o executável, armazenando-o no arquivo <b>a.out</b> .
<b>gcc prog.c aux.o ent_sai.s</b>	Compila o programa cujo código está distribuído nos arquivos <b>prog.c</b> , <b>aux.o</b> e <b>ent_sai.s</b> . Para o código em <b>prog.c</b> todas as etapas são executadas; para o código em <b>ent_sai.s</b> apenas as etapas de montagem e ligação são executadas; e o código em <b>aux.o</b> apenas participa da etapa de ligação, juntamente com os outros dois arquivos-objetos que são gerados ( <b>prog.o</b> e <b>ent_sai.o</b> ). O executável é armazenado no arquivo <b>a.out</b> .
<b>gcc -o prog prog.o</b>	Compila o programa que está no arquivo <b>prog.o</b> , executando apenas a

	etapa de ligação. O executável é armazenado no arquivo <b>prog</b> .
<b>gcc prog.c -o prg_exem aux.o ent_sai.s</b>	Compila o programa cujo código está distribuído nos arquivos <b>prog.c</b> , <b>aux.o</b> e <b>ent_sai.s</b> da forma já descrita. O programa executável é armazenado no arquivo <b>prg_exem</b>

Este exemplo mostra que a opção -o (arqsai) pode aparecer em qualquer posição da linha de comando.

## Compilações Parciais

Nas formas vistas até agora, o comando gcc realiza todas as etapas da compilação, destruindo os arquivos intermediários (código assembler e código-objeto): apenas os arquivos-fontes originais e o arquivo executável permanecem. Entretanto, é possível determinar ao compilador que execute etapas específicas, preservando os arquivos intermediários, com as seguintes opções de compilação:

- E Realiza apenas a etapa de pré-processamento. O texto da unidade de compilação é mostrado no terminal. Pode-se usar a opção -o para armazenar a saída em um arquivo.
- S Realiza as etapas de pré-processamento e compilação. Para cada unidade de compilação é gerado um código assembler que fica armazenado em um arquivo com a extensão .s.
- c Realiza as etapas de pré-processamento, compilação e montagem. Para cada unidade de compilação é gerado um código-objeto que fica armazenado em um arquivo com extensão .o.

## Especificando o padrão a ser usado

Alguns compiladores podem trabalhar com vários padrões para uma mesma linguagem. Esse é o caso do compilador **gcc**, que adota como norma um padrão próprio, estendendo o padrão ISO/IEC 9899:1990. Para fazer o gcc adotar o padrão ISO/IEC 9899:1999, deve-se usar a opção de compilação “**-std=c99**”. (Para versões do compilador que implementam o padrão de 2011, deve-se usar a opção “**-std=c11**”). Recomenda-se também que as compilações sejam realizadas com as opções “**-Wall**”, para **forçar** a produção de todos **os avisos do compilador**, e “**-pedantic**”, para que os **diagnósticos especificados pelo padrão sejam fornecidos**. Os comandos de compilação devem ser da forma:

```
gcc prog.c -o prg_exem aux.o ent_sai.s -std=c11 -Wall -pedantic
```

**Parou em:** Introducing Debugging, na página 46

---

### **Referências:**

#### **C Primer Plus:**

PRATA, Stephen. C Primer Plus. 6. ed. São Paulo: Pearson Education do Brasil, 2014.

#### **C Como Programar (Nova Edição Atualizada):**

DEITEL, Paul; DEITEL, Harvey. C Como Programar (Nova Edição Atualizada). 6. ed. São Paulo: Pearson Education do Brasil, 2014.