

Programação Aplicada à Estatística

Pedro Rafael Diniz Marinho

Universidade Federal da Paraíba

2021.2

Sobre mim

Sou Pedro Rafael D. Marinho e serei o professor de vocês nesse período letivo. Sou Dr. em Estatística. Toda minha formação foi na área de Estatística (bacharelado, mestrado e doutorado).

Sou lotado no Centro de Ciências Exatas e da Natureza no Departamento de Estatística - UFPB.

A minha sala é a de número 12.

Meu email: pedro.rafael.marinho@gmail.com

Sobre mim

Trabalhei no mestrado com **modelos lineares com heteroscedasticidade de forma desconhecida** com orientação do PhD Francisco Cribari Neto. Na dissertação desenvolvemos simulações de bootstrap (simples e duplo) para avaliação da cobertura dos intervalos de confiança que indexam os parâmetros desses modelos. Um pacote na linguagem R foi desenvolvido.

Título da dissertação

Estimadores Intervalares Sob Heteroscedasticidade de Forma Desconhecida Via Bootstrap Duplo

O pacote `hcci` na versão 1.0.0 encontra-se hospedado no CRAN do R em <https://cran.r-project.org/web/packages/hcci/index.htm>.

Sobre mim

No doutorado trabalhei com o PhD Gauss Moutinho Cordeiro na área de **distribuições de probabilidade**. Na tese foram criados novas classes de distribuições de probabilidade em que é possível gerar uma nova distribuição a partir de uma distribuição G conhecida (baseline). Também foi construído o pacote AdequacyModel na linguagem R que encontra-se atualmente na versão 2.0.0 sob os termos da licença $\text{GPL} \geq 2$ (*GNU General Public License*)

Título da tese

Some New Families of Continuous Distributions

O pacote poderá ser obtido em <https://cran.r-project.org/web/packages/AdequacyModel/index.html>.

Ementa

O curso de **Programação Aplicada à Estatística** é formado pela seguinte ementa.

Ementa do Curso

Introdução: Modelo de um computador digital; Linguagem de máquina; Introdução à Programação; Histórico das linguagens de programação; Compiladores e Interpretadores; Lógica e Lógica de Programação; Construção de algoritmos; Pseudocódigo.

Linguagem C: Visão geral; Expressões; Controle de fluxo; Funções; Ponteiros, Vetores e Matrizes; Alocação dinâmica de memória; Cadeias de caracteres; Tipos estruturados; Arquivos; Ordenação e Busca. Aplicações práticas à Estatística.

Plano de Curso

O **plano de curso** seguirá a ementa apresentada logo acima no *frame* anterior.

O plano de curso é de responsabilidade do professor da disciplina e deve estar de acordo com a ementa da disciplina.

O plano de curso é estabelecido pelo professor e ficará a cargo desse a sua construção e alteração, se necessário, no decorrer do curso, desde que o mesmo esteja de acordo com a ementa do curso.

O rigor e profundidade do assunto ficará a cargo do professor e faz parte do seu plano de curso.

Plano adotado no momento

1. Modelo de um computador digital.
2. Elementos básicos de um computador típico.
3. Armazenamento de dados e programas na memória.
4. Linguagem de máquina.
5. Histórico das linguagens de programação.
6. Tipos de linguagens.
7. Compiladores e Interpretadores.
8. Lógica e Lógica de programação
9. Construção de algoritmos.
10. Linguagem C.
11. Variáveis.
12. Operadores.
13. Entrada e Saída.
14. Controle de fluxo.
15. Construções de Laços.
16. Seleção.
17. Funções.
18. Ponteiros, Vetores e Matrizes.
19. Alocação dinâmica de memória.
20. Cadeia de caracteres.
21. Tipos estruturados.
22. Arquivos.
23. Ordenação e busca.
24. Aplicações práticas à Estatística.

Bibliografias Utilizadas

Abaixo estão enumeradas as bibliografias **básicas** utilizadas:

- ① Celes, W., Cerqueira, R., Rangel, J.L. Introdução a Estruturas de Dados: Com Técnicas de Programação em C. Rio de Janeiro: Elsevier, 2004.
- ② Kernighan, B. W., Ritchie, D. M. C: A linguagem de programação padrão ANSI. Rio de Janeiro: Elsevier, 1989.
- ③ Schildt, H. C Completo e Total. São Paulo: Makron Books, 1996.

Bibliografias Utilizadas

Abaixo estão enumeradas as bibliografias **básicas** utilizadas:

- ① Oliveira, U. Programando em C Volume I: Fundamentos. Rio de Janeiro: Ciência Moderna, 2008.
- ② Press, W. H., Teukolsky, S. A., Vetterling, W. T., Flannery, B. P. Numerical recipes in C: the art of scientific computing, Cambridge : Cambridge University Press, 1994.

Recursos utilizados

Boa parte do curso será apoiada pelo uso de **datashow** o que nos ajudará bastante a decorrer sobre os diversos assuntos contidos na ementa desse curso que é bastante ampla.

O quadro será utilizado para resolução de alguns exemplos bem como complementações em que o professor achar conveniente no momento de aula.

Sobre as avaliações

No curso iremos considerar **três avaliações**, duas provas e **TALVEZ** um trabalho.

As duas primeiras avaliações serão provas que irão contemplar o conteúdo ministrado em sala de aula (**tudo que foi dito, apresentado e escrito**).

No caso em que for decidido por um trabalho como terceira avaliação, este deverá ser formado por um grupo de no máximo 3 pessoas.

Sobre as avaliações

As datas e temas dos trabalhos serão fornecidos após a segunda avaliação. A divisão dos grupos ficará a cargo dos alunos.

O professor tomará partido na divisão em caso de problemas.

Os trabalhos serão avaliados segundo sua organização, profundidade do assunto e irei avaliar o código que será fornecido no Apêndice do trabalho e enviado para meu email.

Importante

A reposição do trabalho será uma prova referente ao assunto do tema da equipe em que o aluno encontra-se presente.

Sobre as reposições

O aluno terá direito a apenas uma reposição de uma das avaliações desde que satisfeito o que rege a **Resolução N° 16/2015 que aprova o Regulamento dos Cursos Regulares de Graduação da UFPB.**

O aluno poderá repor uma prova desde que entre com pedido de reposição junto à coordenação do seu curso. O coordenador de seu curso irá avaliar o pedido de reposição com base na **Resolução N° 16/2015** do CONSEPE e encaminhará o pedido julgado ao Departamento de Estatística - UFPB.

Apenas irá repor a prova quem atender os requisitos do Art. 92, 6º §.

Aos alunos interessados (\LaTeX)

Aconselho fortemente ao aluno que pretende produzir textos de qualidade (qualidade tipográfica) considerar o uso da linguagem de comandos macros \LaTeX .

\LaTeX é uma linguagem de comandos macros de \TeX e atualmente encontra-se na versão $\text{\LaTeX} 2\epsilon$.

Com o uso de \LaTeX a facilidade de construir texto de alta qualidade tipográfica será uma ferramenta a mais na mão de um profissional em estatística.

Com \LaTeX é possível fazer grandes mudanças em um texto em poucos minutos apenas acrescentando alguns comandos ao preâmbulo do código.

Aos alunos interessados (\LaTeX)



Figura: Donald Knuth.

\TeX é um sistema de tipografia científica desenvolvido por **Donald E. Knuth** que é orientado à produção de textos técnicos e fórmulas matemáticas. A pedido da AMS (*American Mathematical Society*), Donald Knuth desenvolveu uma linguagem de computador para editoração de textos com muitas equações.

Aos alunos interessados (\LaTeX)



Figura: Donald Knuth.

O trabalho de criação do TeX se estendeu de 1977 a 1998, quando TeX foi disponibilizado gratuitamente. O TeX possui aproximadamente **600 comandos** que controlam a construção de uma página.

Pode-se considerar o TeX como sendo um compilador para textos científicos que produz documentos de alta qualidade tipográfica.

Aos alunos interessados (\LaTeX)

O \TeX atingiu um estado de desenvolvimento em que Beebe (1990 afirmou):

“Meu trabalho no desenvolvimento de \TeX , METAFONT e as fontes Computer Modern chegou ao final. Eu não irei realizar mudanças futuras, exceto corrigir sérios erros de programação.”

Ver em: BEEBE, N. H. Comments on the future of TeX and METAFONT. TUGboat, v. 11, n. 4, p. 490–494, 1990.

Aos alunos interessados (\LaTeX)

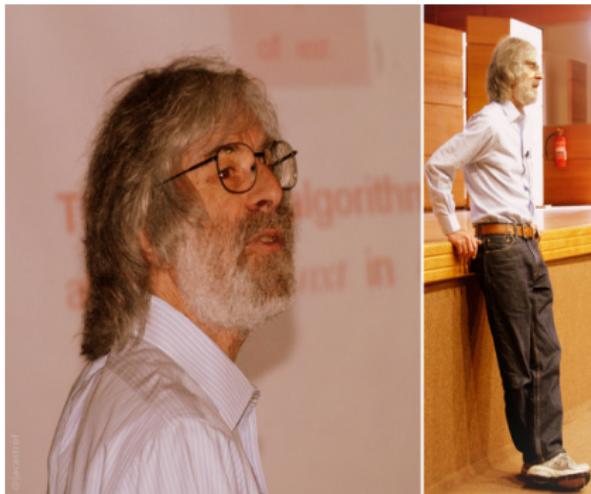


Figura: Leslie Lamport.

Quase que em paralelo foi desenvolvido por Leslie Lamport o \LaTeX . Essas macros definem tipos de documentos, tais como livros, artigos, cartas, entre outros.

Inclusive essa apresentação é um tipo básico de documento que foi produzido em \LaTeX .

Aos alunos interessados (\LaTeX)

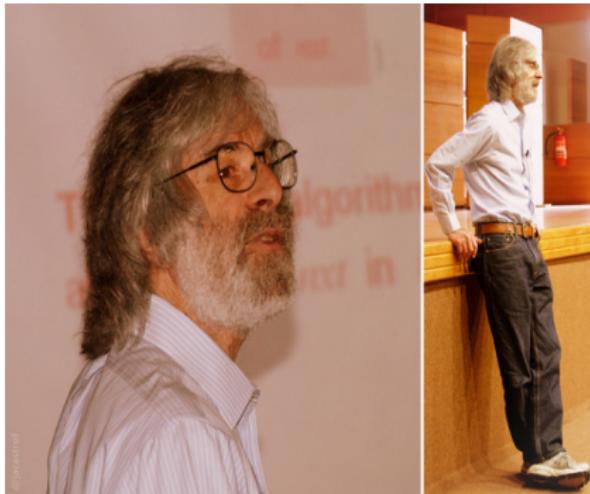


Figura: Leslie Lamport.

Para maiores detalhes leia sobre o pacote `beamer` que está disponível com a maioria das distribuições \LaTeX . Há diversos documentos disponíveis nos mais variados idiomas na rede.

`beamer` também está disponível no **The Comprehensive \TeX Archive Network (CTAN)**.

Aos alunos interessados (\LaTeX)



Figura: CTAN lion drawing
by Duane Bibby.

CTAN é o lugar central para todos os tipos de material em torno de \TeX . CTAN tem atualmente **5752 pacotes** e **2637 colaboradores** contribuíram para essa quantidade de pacotes.

O símbolo ao lado foi desenhado pelo artista comercial Duane Bibby. Este leão foi utilizado nas ilustrações para o livro $\text{\TeX}Book$ de Donald Knuth e apareceu com grande frequência em outros materiais.

Aos alunos interessados (\LaTeX)



Figura: CTAN lion drawing
by Duane Bibby.

Maiores detalhes sobre o CTAN podem ser encontrados em <https://www.ctan.org/lion/>. Desde dezembro de 1994, o pacote \LaTeX está sendo atualizado pela equipe $\text{\LaTeX} 3$, dirigida por Frank Mittelbach, para incluir algumas melhorias que já vinham solicitadas a algum tempo. A equipe se preocupa também em reunificar todas as versões modificadas que surgiram desde o aparecimento do $\text{\LaTeX} 2.09$.

Aos alunos interessados (\LaTeX)



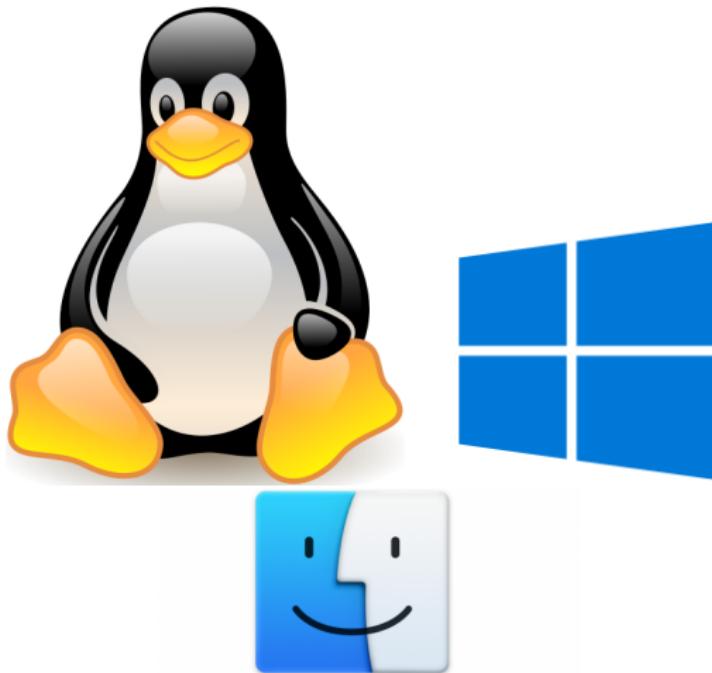
Figura: CTAN lion drawing
by Duane Bibby.

O melhor de tudo, o \LaTeX é um sistema estável mas com crescimento constante, podendo ser instalado em quase todos os sistemas operacionais.

Usuários de Unix, Linux, Windows ou Mac OS X podem dispor de todo ferramental para produzir ótimos textos com o \LaTeX .

Nota: Pronuncia-se “leitec” e não “latéx”.

Aos alunos interessados (\LaTeX)



MacTM OS

Como instalo o L^AT_EX no Linux?

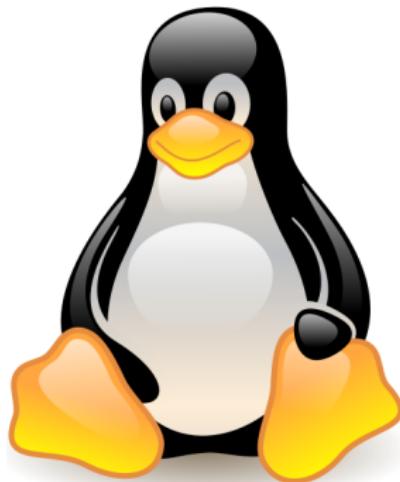


Figura: Tux (Mascote do Linux).

Como instalo o L^AT_EX no Linux?

Inicialmente é preciso instalar o compilador de L^AT_EX. Recomendo o uso do T_EX Live.

A maioria das distribuições linux (Arch, Ubuntu, Fedora, Mint, Sabayon, entre outros “sabores”) apresentam esse compilador de L^AT_EX em seus repositórios.

Por exemplo, no **Arch Linux** e distribuições derivadas que utilizam os mesmos repositórios do Arch como Antergos façam:

```
sudo pacman -S texlive.
```

Como instalo o L^AT_EX no Linux?

No **Ubuntu** ou qualquer distribuição que faz uso dos repositórios do Ubuntu façam:

```
sudo apt-get install texlive-full.
```

Já os usuários da distribuição **Fedora** e distribuições derivadas que utilizam-se dos mesmos repositórios devem fazer:

```
sudo dnf -y texlive-scheme-full.
```

Observação: Todos os comandos acima devem ser executados no terminal da respectiva distribuição com permissão de super usuário (usuário que pode fazer alterações no sistema operacional).

Como instalo o L^AT_EX no Windows?



Como instalo o L^AT_EX no Windows?

Felizmente, há o T_EXLive para Windows que poderá ser obtido no site oficial do projeto T_EXLive.

O usuário de Windows deverá baixar o arquivo

`install-tl-windows.exe`

que possui aproximadamente **13mb**.

Nota: `install-tl-windows.exe` é apenas o instalador do T_EXLive para Windows. Dessa forma, ao final da instalação, o T_EXLive terá muito mais que apenas 13mb instalado em seu computador.

Como instalo o L^AT_EX no Linux?

Mas para escrevermos um texto com qualidade usando o L^AT_EX precisamos também de um editor de texto.

Na maioria dos casos usamos uma **IDE** (*Integrated Development Environment*) (**Ambiente de Desenvolvimento Integrado**)

Aconselho o uso do T_EXstudio que está disponível para Linux, Windows e Mac OS.

O T_EXstudio é um software sobre os termos da licença **GPL** (*GNU General Public License*) e pode ser obtido em
<http://texstudio.sourceforge.net/>.

TEXstudio

Observação: Aperte F6 para compilar o documento e F7 para visualizar o PDF produzido por meio de código LATEX.

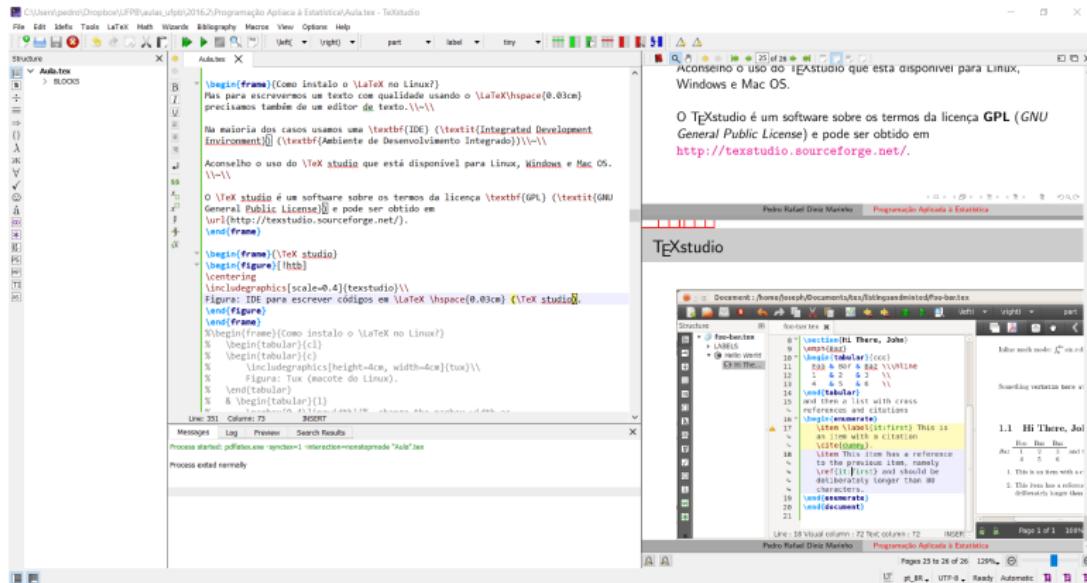


Figura: IDE para escrever códigos em LATEX (TeXstudio).

Vantagens do TEXstudio

Vantagens do TEXstudio

- ① É uma IDE leve.

Vantagens do TEXstudio

- ① É uma IDE leve.
- ② O TEXstudio auto-completa os comandos que vão sendo digitados.

Vantagens do TEXstudio

- ① É uma IDE leve.
- ② O TEXstudio auto-completa os comandos que vão sendo digitados.
- ③ É possível ir de um ponto específico do código LATEX para o ponto correspondente no PDF criado.

Vantagens do TEXstudio

- ① É uma IDE leve.
- ② O TEXstudio auto-completa os comandos que vão sendo digitados.
- ③ É possível ir de um ponto específico do código LATEX para o ponto correspondente no PDF criado.
- ④ Tem licença **GPL** (*GNU General Public License*), isto é, não pago por ela e tenho acesso ao seu código fonte.

Vantagens do TEXstudio

- ① É uma IDE leve.
- ② O TEXstudio auto-completa os comandos que vão sendo digitados.
- ③ É possível ir de um ponto específico do código LATEX para o ponto correspondente no PDF criado.
- ④ Tem licença **GPL** (*GNU General Public License*), isto é, não pago por ela e tenho acesso ao seu código fonte.
- ⑤ O PDF é visualizado ao lado do código LATEX.

Como aprender L^AT_EX?

Pergunta do aluno

Okay professor, o senhor me convenceu em utilizar L^AT_EX. Então, como eu posso aprender os seus comandos? Há algum material interessante para se começar a estudar L^AT_EX?

Resposta do professor legal

Há vários livros e materiais disponibilizados na internet. Há diversos grupos de discussão sobre L^AT_EX. Para não complicar muito, aconselho estudar por essa apostila:

http://www.univasf.edu.br/~joseamerico.moura/index_arquivos/lenimar_tex_LATEX.pdf

Programação

O que é programação?

Resposta: Linguagem de programação é um método padronizado para comunicar instruções para um computador por meio de uma sintaxe.

Trata-se de um conjunto de regras sintáticas utilizadas para passar instruções para um computador. Por meio dessas regras, é possível que o programador especifique os **tipos de dados** em que o professor irá processar.

Tais dados serão armazenados e transmitidos e/ou transmitidos entre os componentes que formam o computador. Assim, a linguagem também permite especificar quais ações devem ser tomadas e sob quais circunstâncias serão tomadas.

Linguagens



Figura: Diversas linguagens de programação disponíveis para uso.

Por que aprender à programar?

Por que um estatístico deve saber programar?

Por que aprender à programar?

Por que o estatístico deve saber programar?

Resposta: Simplesmente pelo fato de que **não** dá para fazer muita coisa na estatística (principalmente no mercado de trabalho) se o profissional não é capaz de fazer com que o computador resolva os seus problemas.

Um aluno questiona...

Mas professor, temos o SPSS, Excel, SAS, Statistica e outros softwares em que podemos chamar nosso conjunto de dados e apertar centenas de botões e ter **alguns** resultados.

Por que aprender à programar?

O professor continua respondendo...

A maioria desses software não possuem as técnicas estatísticas mais recentes e muitas vezes não são capazes de se adequar aos problemas específicos que nos deparamos ao tentar resolver um problema.

Muitas vezes precisamos modificar uma função programada por uma outro programador para que ela venha a funcionar no nosso problema.

Observação: Diversas outras vezes precisamos programar para realizar simulações. É muito comum na estatística estudar propriedades de algumas estatística ou modelo estatístico e querer simular o seu comportamento em diversos cenários diferentes.

Por que aprender à programar?

Uma linguagem de programação bastante utilizada na estatística é a linguagem R.

Linguagem R

R é uma linguagem de programação para computação estatística e gráficos. R é uma parte oficial do projeto GNU da Free Software Foundation's.

Curiosidade: A linguagem R foi criada originalmente por Ross Ihaka e Robert Gentleman no **Departamento de Estatística** da Universidade de Auckland, Nova Zelândia em agosto de 1993.

Nota: É muito importante que um estatístico saiba programar na linguagem R. Alguns empregos exigem isso. Porém, se não exigirem, o R te ajudará bastante.

Por que aprender à programar?

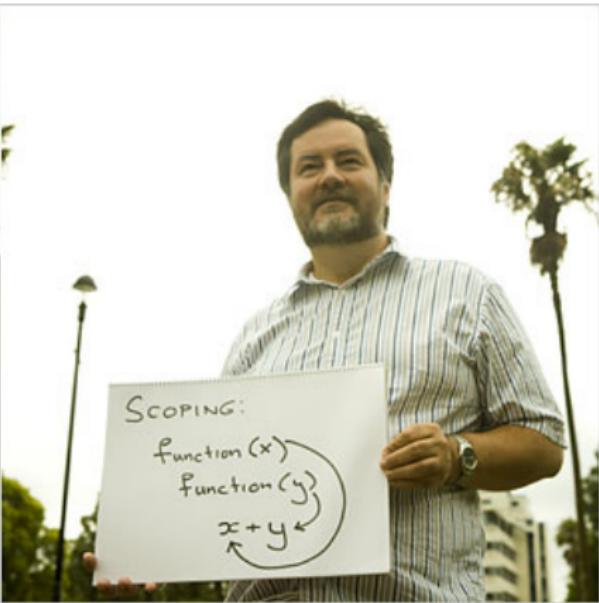
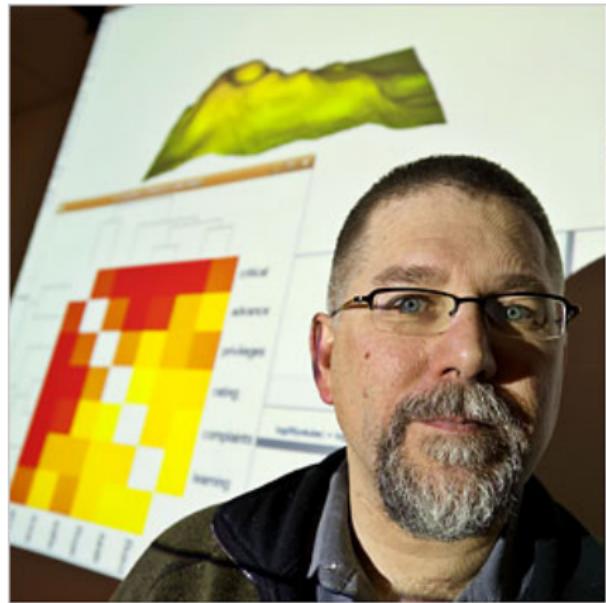


Figura: Criadores da linguagem R [Robert Gentleman (foto à esquerda) e Ross Ihaka (foto à direita)].

Por que aprender à programar?



Figura: Logo da linguagem R.

Um dos grandes motivos da grande popularidade da linguagem R se deve a grande quantidade de pacotes disponíveis para os usuários da linguagem.

Atualmente há mais de 5 mil pacotes para R com o foco nas mais variadas áreas: estatística, matemática, biologia, economia, entre outras.

Por que aprender à programar?



Figura: Logo da linguagem R.
Obtenha a linguagem R em
<https://www.r-project.org/>.

Observação: Para programar em R **não é suficiente entender alguns pacotes específicos**. É preciso entender a sintaxe base da linguagem que nos permite inclusive criar outros pacotes e melhorar os existentes.

Por que aprender à programar?



Figura: Logo da linguagem R.
Obtenha a linguagem R em
<https://www.r-project.org/>.

Observação: As novas metodologias estatísticas chegam mais rapidamente em R do que em outros softwares estatísticos pelo fato do R ser uma linguagem livre (código aberto e gratuita).

Esclarecedor

“Ciência
da computação tem tanto
a ver com o computador como
a Astronomia com o telescópio,
a Biologia com o microscópio,
ou a Química com os
tubos de ensaio. A Ciência não
estuda ferramentas, mas o que
fazemos e o que descobrimos
com elas.” - **Edsger Dijkstra**
(Prêmio Turing em 1972)



Figure: Edsger Dijkstra

Máquina de Turing

A máquina de Turing é um dispositivo teórico conhecido como máquina universal, que foi concebido pelo matemático britânico Alan Turing (1912-1954), muitos anos antes de existirem os modernos computadores digitais.

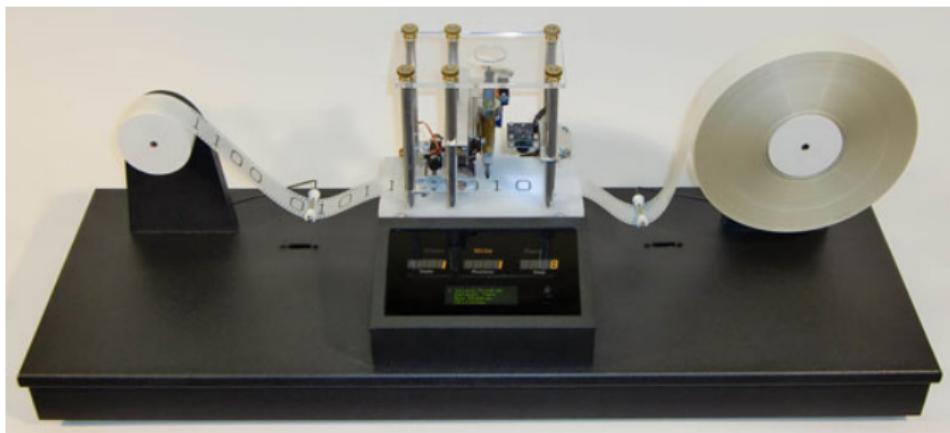


Figure: Exemplo de uma Turing física.

Máquina de Turing

Na Teoria da Computabilidade, um problema é solúvel se há uma Máquina de Turing para aquele problema

Em seu artigo original, Turing demonstra a existência de um **problema insolúvel**.

Existe basicamente dois tipos de máquinas de Turing:

- ① Máquina de Turing **Determinística**: (Se 'A', então 'B')
- ② Máquina de Turing **Não-Determinística**: (Se 'A', então 'B' **ou** 'C' **ou** 'D' **ou** ...)

Teoria da Computabilidade

A computabilidade é a Teoria da Complexidade Computacional que estudam os **limites** da computação:

- ① Quais problemas jamais poderão ser resolvidos por um computador, independente de sua velocidade ou memória?
- ② Quais problemas podem ser resolvidos por um computador, mas requerem um período tão extenso de tempo para completar a ponto de tornar a solução impraticável?
- ③ Em que situações podem ser mais difícil resolver um problema do que verificar cada uma das soluções?

Teoria da Computabilidade

Das três perguntas anteriores, a última é referente às **classes de paradigmas**, são elas:

- Classe **P**: De tempo polinomial determinístico. Os algoritmos pertencentes à esta classe são chamados de **algoritmos eficientes**.
- Classe **NP**: De tempo polinomial não-determinístico.

O conjunto de problemas que não podem ter solução em tempo polinomial mas candidatos a solução podem ser checados em tempo polinomial são problemas pertencentes à classe NP.

Somente uma máquina de Turing não-determinística podem resolver esses problemas. Eles são resolvidos em tempo polinomial por uma máquina de Turing não-determinística que acerta em todos os passos.

Teoria da Computabilidade

Na matemática, a questão a respeito de $P = NP$ ou $P \neq NP$ é um problema em aberto.

A grande importância dessa classe (NP) de problemas se baseia no fato de que ela contém muitos problemas de busca e otimização para os quais gostaríamos de saber se há uma solução.

Exemplos: Problema do Caixeiro Viajante, problema da mochila, entre outros.

E se $P = NP$?



- ① Diga adeus a criptografia;
- ② Soluções matemáticas não complicadas;
- ③ Previsão do tempo, terremotos e tsunamis;

Problemas NP (precisa-se de linguagens eficientes)

Problemas NP-Completos fazem parte de nossas vidas...

Na estatística sempre nos deparamos com problemas NP-Completos, isto é, sempre lidamos com problemas que não possuem soluções em tempo polinomial.

Exemplo: Constantemente precisamos estimar parâmetros de um modelo probabilístico por meio do método de máxima verossimilhança, isto é, maximizamos a função de log-verossimilhança de um modelo probabilístico. A otimização global é um problema NP-Completo.

Observação: Atualmente está cada vez mais complicados realizar tais otimizações uma vez que os modelos cada vez mais estão adicionando parâmetros extras o que torna a função extremamente complicada em alguns casos.

Breve História da Linguagem C

Embora possua um nome estranho quando comparada com outras linguagens de programação da terceira geração, como FORTRAN, PASCAL, ou COBOL, a linguagem C é uma das linguagens mais importantes até hoje criada.

Curiosidade: O nome da linguagem (e a própria linguagem) resulta da evolução de uma outra linguagem de programação, desenvolvida pelo programador **Ken Thompson** nos Laboratórios Bell, chamada de B.

Breve História da Linguagem C



Figura: Ken Thompson (sentado) jogando xadrez com um colega.

B trata-se de uma simplificação da linguagem BCPL (*Basic Combined Programming Language*). Assim como BCPL, B só possuía um tipo de dados.

A linguagem B também foi recebida contribuições do Dennis Ritchie (criador de C).

Breve História da Linguagem C



Figura: Dennis Ritchie (criador da linguagem C).

A linguagem C foi criada em 1972 nos *Bell Telephone Laboratories* por Dennis Ritchie com a finalidade de permitir a escrita do sistema operacional Unix.

Desejava-se uma linguagem de alto nível de modo a evitar o uso do Assembly.

Breve História da Linguagem C

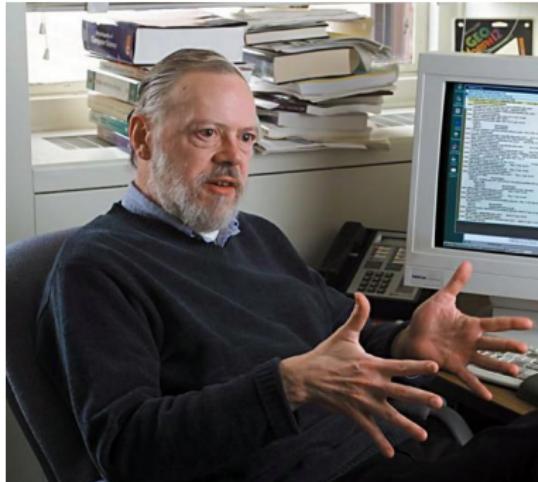


Figura: Dennis Ritchie (criador da linguagem C).

Devido às capacidades e através da divulgação do sistema Unix pelas universidades dos Estados Unidos, a linguagem C deixou cedo as portas dos laboratórios Bell.

C disseminou-se e tornou-se conhecida por todos os tipos de programadores, independentemente dos projetos em que estivessem envolvidos.

Breve História da Linguagem C

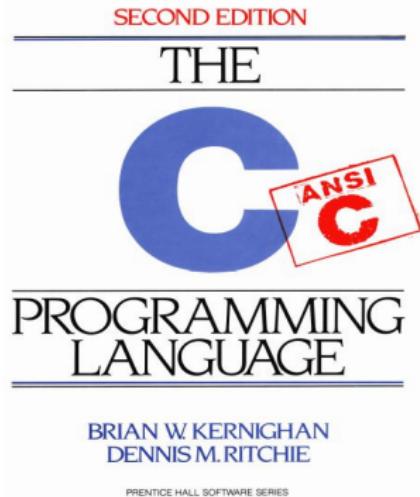


Figura: Ótimo livro sobre a linguagem C.

Essa dispersão de diferentes projetos utilizando a linguagem C levou a que diferentes organizações desenvolvessem e utilizassem diferentes versões da linguagem C criando assim alguns problemas de compatibilidade, entre diversos outros.

O material ao lado é um livro sobre a linguagem C escrito pelo seu criador.

Breve História da Linguagem C

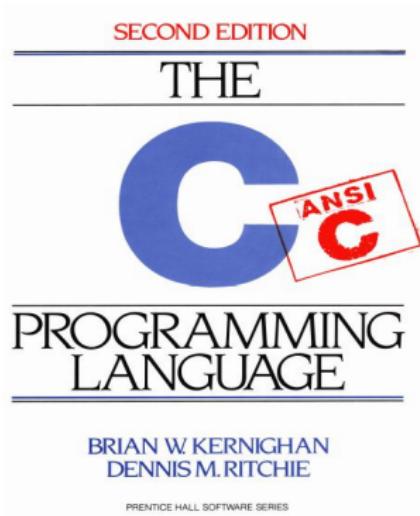


Figura: Ótimo livro sobre a linguagem C.

Devido ao fenômeno que foi a linguagem C e aos problemas de compatibilidade que existiam na época, o **American National Standards Institute** (ANSI) formou em 1983 um comitê para a definição de um padrão para a linguagem C.

Breve História da Linguagem C

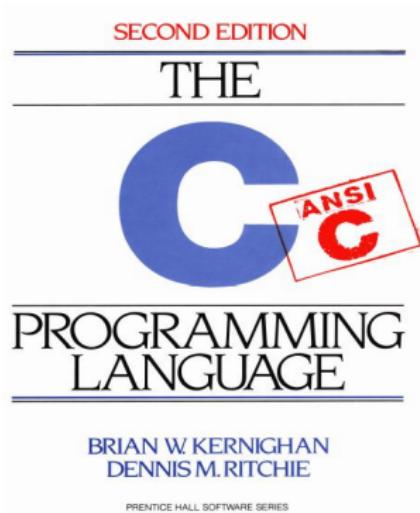


Figura: Ótimo livro sobre a linguagem C.

O padrão tem como objetivo o funcionamento semelhante de todos os compiladores da linguagem, com especificações muito precisas sobre aquilo que a linguagem deve ou não fazer, seus limites, definições, dentre outras coisas.

Mil e Uma Razões para Programar em C

Devido à enorme quantidade de linguagens de programação disponíveis no mercado, seria necessário que uma delas se destacasse muito em relação às outras para conseguir interessar tantos programadores.

A maior parte das linguagens de programação tem um objetivo específico a atingir:

- PASCAL - Ensino de Técnicas de Programação.
- FORTRAN - Cálculo Científico.
- LISP e PROLOG - Vocacionadas para as áreas de Inteligência Artificial.

Mil e Uma Razões para Programar em C

Pergunta do aluno

Certo, entendi, temos linguagens que se destinam a um objetivo específico como as que foram listadas a cima. Mas quanto à C, a que área de desenvolvimento se destina?

Mil e Uma Razões para Programar em C

Pergunta do aluno

Certo, entendi, temos linguagens que se destinam a um objetivo específico como as que foram listadas a cima. Mas quanto à C, a que área de desenvolvimento se destina?

Resposta do professor legal

NENHUMA 😊 .

Por que C?

Qual o porquê de se aprender a linguagem C

C é uma linguagem bastante interessante devido ser uma linguagem de programação de propósito geral, isto é, C pode ser utilizada para os mais variados fins.

Alguns outros motivos para se aprender C são:

Por que C?

Qual o porquê de se aprender a linguagem C

C é uma linguagem bastante interessante devido ser uma linguagem de programação de propósito geral, isto é, C pode ser utilizada para os mais variados fins.

Alguns outros motivos para se aprender C são:

- 1 C é uma linguagem extremamente eficiente.

Por que C?

Qual o porquê de se aprender a linguagem C

C é uma linguagem bastante interessante devido ser uma linguagem de programação de propósito geral, isto é, C pode ser utilizada para os mais variados fins.

Alguns outros motivos para se aprender C são:

- ① C é uma linguagem extremamente eficiente.
- ② Diversas outras linguagem foram programadas utilizando C.
Por exemplo a linguagem Python, R, entre outras.

Por que C?

Qual o porquê de se aprender a linguagem C

C é uma linguagem bastante interessante devido ser uma linguagem de programação de propósito geral, isto é, C pode ser utilizada para os mais variados fins.

Alguns outros motivos para se aprender C são:

- ① C é uma linguagem extremamente eficiente.
- ② Diversas outras linguagem foram programadas utilizando C. Por exemplo a linguagem Python, R, entre outras.
- ③ Na estatística muitas vezes precisamos fazer simulações que são computacionalmente intensivas.

Por que C?

Qual o porquê de se aprender a linguagem C

C é uma linguagem bastante interessante devido ser uma linguagem de programação de propósito geral, isto é, C pode ser utilizada para os mais variados fins.

Alguns outros motivos para se aprender C são:

- ① C é uma linguagem extremamente eficiente.
- ② Diversas outras linguagem foram programadas utilizando C. Por exemplo a linguagem Python, R, entre outras.
- ③ Na estatística muitas vezes precisamos fazer simulações que são computacionalmente intensivas.
- ④ C é uma ótima linguagem para se começar a programar.

Por que C?

Qual o porquê de se aprender a linguagem C

C é uma linguagem bastante interessante devido ser uma linguagem de programação de propósito geral, isto é, C pode ser utilizada para os mais variados fins.

Alguns outros motivos para se aprender C são:

- ① C é uma linguagem extremamente eficiente.
- ② Diversas outras linguagem foram programadas utilizando C. Por exemplo a linguagem Python, R, entre outras.
- ③ Na estatística muitas vezes precisamos fazer simulações que são computacionalmente intensivas.
- ④ C é uma ótima linguagem para se começar a programar.
- ⑤ Códigos em C podem ser importados para R, isto é, R “conversa” com C.

Por que C?

Observação: Pelo fato de C ser uma linguagem de propósito geral (*general purpose*), esta linguagem pode ser utilizada nos mais variados fins, como sistemas operacionais, interfaces gráficas, etc.

Importante

Há uma falácia de que C é uma linguagem extremamente difícil. Na verdade ocorre que muitas pessoas começam a estudar programação por meio de C, momento este em que é somado dificuldades em aprender à programar (lógica de programação) com as dificuldades de se aprender uma sintaxe de uma linguagem de programação.

Observação: Porém, é verdade que muitas coisas temos que fazer nós mesmo em C. Por isso que ela é a linguagem adotada na maioria dos cursos de introdução à programação espalhados pelo mundo.

Por que C?

Outras razões para se utilizar C

C é utilizado quando a velocidade, espaço e portabilidade são importantes. A maioria dos sistemas operacionais das outras linguagens e de grande parte dos softwares e games são escritas em C.

Observação: Há basicamente três padrões de C que podem ser encontrados por aí. São eles:

- ① **ANSI C** que é do fim dos anos de 1980 e é utilizado para códigos mais antigos;
- ② Muitas coisas foram consertadas no padrão C99 de 1999;
- ③ Algumas novidades foram acrescentadas no atual padrão C11 lançado em 2011.

Observação: Não existem grandes diferenças entre as versões de C. Iremos destacá-las ao longo do caminho.

Arquitetura de von Neumann



Figura: John von Neumann.

A arquitetura de von Neumann é uma arquitetura de um computador digital que possibilita uma máquina digital armazenar os seus programas no mesmo espaço de memória que os dados, podendo assim manipular tais programas.

“Não há sentido em ser preciso quando não se sabe sobre o que está a falar” - von Neumann.

Arquitetura de von Neumann

A máquina proposta por von Neumann possui as seguintes componentes:

- ① Uma **memória**;
- ② Uma **unidade aritmética e lógica**;
- ③ Uma **unidade central de processamento** (CPU), composto por diversos registradores;
- ④ Uma **unidade de controle**, cuja função é a mesma da tabela de controle de uma Máquina de Turing universal (estabelece as mudanças de estado por meio das entradas).

Compilador

O que é um compilador?

Compilador

O que é um compilador?

Resposta: Um compilador é um programa de computador ou mesmo um grupo de programas que é responsável por traduzir um código fonte escrito em uma linguagem compilada à um programa equivalente do ponto de vista semântico.

Compilador

O que é um compilador?

Resposta: Um compilador é um programa de computador ou mesmo um grupo de programas que é responsável por traduzir um código fonte escrito em uma linguagem compilada à um programa equivalente do ponto de vista semântico.

O compilador traduz o código fonte de uma linguagem de programação de médio/alto nível para uma linguagem de programação de baixo nível (a exemplo da linguagem Assembly ou código de máquina).

Compilador

Bytecode?

Bytecode?

Alguns compiladores traduzem o código para um formato intermediário, denominado de **bytecode** que é um código de baixo nível. Sendo assim, o bytecode não é imediatamente um arquivo executável.

Bytecode?

Alguns compiladores traduzem o código para um formato intermediário, denominado de **bytecode** que é um código de baixo nível. Sendo assim, o bytecode não é imediatamente um arquivo executável.

Observação: Chamamos de linguagem de baixo nível as linguagens que trabalhamo próximo ao hardware. Baixo nível, médio nível ou alto nível em nada tem a ver com a qualidade da linguagem de programação.

Bytecode?

Alguns compiladores traduzem o código para um formato intermediário, denominado de **bytecode** que é um código de baixo nível. Sendo assim, o bytecode não é imediatamente um arquivo executável.

Observação: Chamamos de linguagem de baixo nível as linguagens que trabalhamo próximo ao hardware. Baixo nível, médio nível ou alto nível em nada tem a ver com a qualidade da linguagem de programação.

Importante: Jamais confunda bytecode com código de máquina. Bytecode é um formato intermediário que irá ser interpretado em uma máquina virtual que fará a execução.

Compilador

A vantagem do bytecode é que o código torna-se mais **portável**, isto é, podemos com o resultado da compilação executar o código proveniente de um processo de compilação em diversas arquiteturas distintas. Dessa forma, o bytecode irá produzir o mesmo resultado esperado em qualquer arquitetura que possua uma máquina virtual que execute o código intermediário.

Exemplos de linguagem que converte o código fonte para bytecode: Java que corre o código sobre a máquina virtual Java, .NET que corre o código sobre a *Common Language Runtime*.

Código Objeto e Código de Máquina

O **código de máquina** é um código binário (0 e 1) que poderá ser executado diretamente pela CPU.

Se abrirmos um arquivo de código de máquina em um editor de texto, veríamos um emaranhado de caracteres sem sentido. É possível ter acesso ao código de máquina em formato hexadecimal por meio de softwares adequados.

O **código objeto** é a saída de um processo de compilação e trata-se de uma parte do código de máquina que ainda não foi vinculado em um programa completo por meio de um **linker**.

Abrindo um Código de Maquina

Ao tentarmos abrir um código de máquina em um editor de texto comum visualizamos algo sem sentido como a sequência de caracteres abaixo:

Abrindo um Código de Maquina

Ao tentarmos abrir um código de máquina em um editor de texto comum visualizamos algo sem sentido como a sequência de caracteres abaixo:

```
MZÀ? $Pÿv?èŠÿ]Ë3ÀP, ?F?ë?fF??, ?< uè2Àëä?Àt?Ba
Àu?C†à2Àùä?¬I, "t??"<\u?€<"u?¬I?öÃ□□é?îY?Ê. <å‰.
?€?~?ä?‰v, ?vüÿv?ÿv?□?èÅ?fÄ?ÿvþÿvü?èüêYY< V?< F
|?ë?Rÿvþÿvü?èWífÄ?< å]ËU< ifìHVW< ~?< F
```

Porém, é possível ter acesso ao código de máquina utilizando editores próprios que apresentam o código em hexadecimal, como o exemplo que segue no *frame* seguinte.

Compilador

```
C:\Utility>debug v.exe  
-d 0 100  
0E3D:0000 CD 20 FF 9F 00 9A F0 FE .....0.....  
0E3D:0010 F0 07 17 03 F0 07 DF 07 .....  
0E3D:0020 FF FF FF FF FF FF FF .....L.  
0E3D:0030 D0 0C 14 00 18 00 3D 0E .....=.....  
0E3D:0040 05 00 00 00 00 00 00 00 .....  
0E3D:0050 CD 21 CB 00 00 00 00 00 ..!.....  
0E3D:0060 20 20 20 20 20 20 20 20 .....  
0E3D:0070 20 20 20 20 20 20 20 20 .....  
0E3D:0080 00 0D 76 2E 65 78 65 0D ..v.exe.D0WS\sys  
0E3D:0090 74 65 6D 33 32 5C 64 6F tem32\dosx..da r  
0E3D:00A0 65 64 65 20 28 63 61 72 ede (carregar an  
0E3D:00B0 74 65 73 20 64 6F 20 64 tes do dosx.exe)
```

Compilador

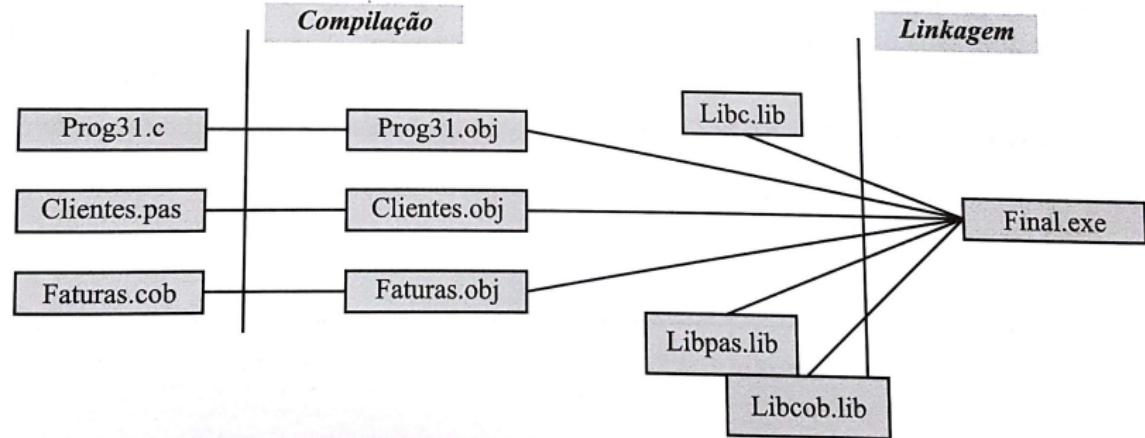


Figura: Diagrama (funcionamento de uma compilador).

Compilador

Alguns autores citam linguagens compiladas em que a tradução do código gera código em C.

Maiores detalhes na referência abaixo

Cooper, Torczon. Engineering a Compiler (em inglês). San Francisco: Morgan Kaufmann, 2003. p. 2. ISBN 1-55860-698-X.

Compilador

Importante

É importante não confundir **compilador** com **tradutor** ou **filtro** que também pode ser chamado de **conversor de linguagem**.

O conversor de linguagem é responsável por converter o código de uma linguagem de alto nível para o código de um outra linguagem de médio/alto nível.

Observação: Um programa que traduz uma linguagem de programação de baixo nível para uma linguagem de programação de alto nível é denominado de **descompilador**.

Compilador



Figura: Grace Hopper.

O primeiro compilador foi escrito por Grace Hopper no ano de 1952 para a linguagem de programação A-0.

Grace Hopper foi analista de sistemas da Marinha dos Estados Unidos. Ela também criou o primeiro compilador para a linguagem COBOL.

Curiosidade: É atribuído à Grace Hopper o termo **bug** utilizado para designar uma falha no código fonte.

Compilador



Figura: Grace Hopper.

Grace Hopper é graduada em matemática e física em 1928 e em 1930 concluiu seu mestrado na Yale University. Em 1934, na mesma Universidade, ela obteve o seu PhD em matemática.

Compilador

Muitos compiladores incluem um **pré-processador** que é um programa separado mas invocado pelo compilador antes do início do processo de tradução.

Normalmente é pre-processador responsável por mudanças no código fonte destinadas de acordo com decisões tomadas em tempo de compilação.

Em programas em C há diversas diretivas para inclusão de novos códigos disponíveis em bibliotecas ou código a parte escrito pelo programador que é informado sua existência por meio de diretivas para o pre-processador.

Compilador

Exemplo: É o pré-processador que substitui os comentários do código fonte por espaços em branco. Ou seja, o compilador não “enxerga” nenhum comentário.

Exemplos de linguagens compiladas: C, C++, Fortran, Object-C, Ocaml, BASIC, COBOL, Ada, D, entre outras.

O que é um interpretador?

Resposta: Interpretadores são programas de computador que tem a finalidade de ler o código-fonte de uma linguagem de programação interpretada e o converte em um código executável.

O que é um interpretador?

Resposta: Interpretadores são programas de computador que tem a finalidade de ler o código-fonte de uma linguagem de programação interpretada e o converte em um código executável.

Seu funcionamento pode ser, em geral, de duas formas:

- O interpretador lê linha-por-linha e converte o código fonte em código objeto (ou bytecode) a medida que vai executando o programa.

Interpretador

O que é um interpretador?

Resposta: Interpretadores são programas de computador que tem a finalidade de ler o código-fonte de uma linguagem de programação interpretada e o converte em um código executável.

Seu funcionamento pode ser, em geral, de duas formas:

- O interpretador lê linha-por-linha e converte o código fonte em código objeto (ou bytecode) a medida que vai executando o programa.
- O interpretador converte o código fonte por inteiro e depois o executa.

O que é um interpretador?

Resposta: Interpretadores são programas de computador que tem a finalidade de ler o código-fonte de uma linguagem de programação interpretada e o converte em um código executável.

Seu funcionamento pode ser, em geral, de duas formas:

- O interpretador lê linha-por-linha e converte o código fonte em código objeto (ou bytecode) a medida que vai executando o programa.
- O interpretador converte o código fonte por inteiro e depois o executa.

Exemplos de linguagens interpretadas: R, Perl, Python, Haskell, Lua, Ruby, Lisp, JavaScript, entre outras.

Programando em C

Na linguagem C existe uma função em que são colocadas todas as instruções que queremos que sejam executadas. Essa função chama-se `main()`, e todo código a executar é colocado entre `{}`.

Ao conjunto de código existente dentro de `{}` chamaremos de **bloco**.

Exemplo: Compile e execute seu primeiro programa utilizando a IDE Code::Blocks.

```
main()
{
}
```

Programando em C

Como podemos observar, este programa faz uma das coisas que mais gosto na vida: não fazer nada. 😊

Programando em C

Como podemos observar, este programa faz uma das coisas que mais gosto na vida: não fazer nada. 😊

Observe com atenção cada linha do programa. A primeira linha é composta pela palavra chave `main` que delimita o local onde todos os programas em C começam.

Programando em C

Como podemos observar, este programa faz uma das coisas que mais gosto na vida: não fazer nada. 😊

Observe com atenção cada linha do programa. A primeira linha é composta pela palavra chave `main` que delimita o local onde todos os programas em C começam.

Observação: Para indicar que `main` é uma função, tal palavra chave é seguida com parênteses - `main()` pois **em C qualquer função tem que ser seguida por parênteses**.

Importante

Os parênteses sem mais nada após o nome da função indicam que a função não recebe qualquer informação do mundo exterior.

Programando em C

Importante

Os parênteses sem mais nada após o nome da função indicam que a função não recebe qualquer informação do mundo exterior.

Mais Importante Ainda

C é uma linguagem **Case Sensitive**, isto significa que C faz diferenciação entre maiúscula e minúscula. Assim, não são a mesma coisa escrever `main()`, `Main()`, `MAIN()`, `mAiN()`, etc.

Programando em C

Importante

Os parênteses sem mais nada após o nome da função indicam que a função não recebe qualquer informação do mundo exterior.

Mais Importante Ainda

C é uma linguagem **Case Sensitive**, isto significa que C faz diferenciação entre maiúscula e minúscula. Assim, não são a mesma coisa escrever `main()`, `Main()`, `MAIN()`, `mAiN()`, etc.

Dica: Todas as funções em C são escritas em letra minúscula, e só se deve utilizar letras maiúsculas quando desejamos utilizar variáveis, mensagens ou funções escritas por nós.

Programando em C

No exemplo anterior, o código não executa nada uma vez que não há código dentro do bloco de instruções composto pela função `main()`.

Nota: É possível que o seu compilador tenha apresentado um *WARNING* com uma mensagem semelhante a “**Function should return value**” ou “**main: no return value**”. Isso se deve ao fato de que a princípio estamos utilizando um compilador para C++.
Duas possíveis soluções caso isso ocorra são:

Programando em C

No exemplo anterior, o código não executa nada uma vez que não há código dentro do bloco de instruções composto pela função `main()`.

Nota: É possível que o seu compilador tenha apresentado um *WARNING* com uma mensagem semelhante a “**Function should return value**” ou “**main: no return value**”. Isso se deve ao fato de que a princípio estamos utilizando um compilador para C++.
Duas possíveis soluções caso isso ocorra são:

- ① Ignorar o aviso. *WARNING* não é erro mas devemos prestar atenção nos avisos para evitarmos problemas.

Programando em C

No exemplo anterior, o código não executa nada uma vez que não há código dentro do bloco de instruções composto pela função `main()`.

Nota: É possível que o seu compilador tenha apresentado um *WARNING* com uma mensagem semelhante a “**Function should return value**” ou “**main: no return value**”. Isso se deve ao fato de que a princípio estamos utilizando um compilador para C++.
Duas possíveis soluções caso isso ocorra são:

- ① Ignorar o aviso. *WARNING* não é erro mas devemos prestar atenção nos avisos para evitarmos problemas.
- ② Coloque a palavra chave `void` antes da função `main()` na forma que segue.

Programando em C

```
void main()
{
}
```

Programando em C

```
void main()
{
}
```

Mais tarde nós saberemos o porquê do uso de void.

Programando em C

```
void main()
{
}
```

Mais tarde nós saberemos o porquê do uso de void.

Vamos agora escrever um novo programa que faz algo mais interessante do que nada. Considere o exemplo que segue:

Programando em C

Exemplo:

```
1: #include <stdio.h>
2: main()
3: {
4:     printf("Hello Mundo Cruel");
5: }
```

Programando em C

Exemplo:

```
1: #include <stdio.h>
2: main()
3: {
4:     printf("Hello Mundo Cruel");
5: }
```

Esse código é em tudo semelhante ao anterior, com a exceção da existência de uma linha de código entre as chaves.

Programando em C

Exemplo:

```
1: #include <stdio.h>
2: main()
3: {
4:     printf("Hello Mundo Cruel");
5: }
```

Esse código é em tudo semelhante ao anterior, com a exceção da existência de uma linha de código entre as chaves.

A linha 4 é responsável pela apresentação da mensagem que desejamos imprimir: “Hello Mundo Cruel”.

Importante

Sempre que queremos tratar conjunto de caracteres temos que colocá-los entre aspas, para que sejam considerados como um todo, isto é, devemos fazer "Hello Mundo Cruel" para o exemplo acima.

Programando em C

Importante

Sempre que queremos tratar conjunto de caracteres temos que colocá-los entre aspas, para que sejam considerados como um todo, isto é, devemos fazer "Hello Mundo Cruel" para o exemplo acima.

A função printf()

Uma das funções que permite a escrita na tela é a função `printf()` = `print` + `formatado`. Como trata-se de uma função, há a necessidade de colocar os parênteses.

Importante

Sempre que queremos tratar conjunto de caracteres temos que colocá-los entre aspas, para que sejam considerados como um todo, isto é, devemos fazer "Hello Mundo Cruel" para o exemplo acima.

A função printf()

Uma das funções que permite a escrita na tela é a função `printf()` = `print` + `formatado`. Como trata-se de uma função, há a necessidade de colocar os parênteses.

Dentro do parênteses é feita a comunicação com a função. Nesse exemplo, passamos a **string** (cadeia de caracteres) que queremos que seja escrita - `printf("Hello Mundo Cruel")`.

Muito Importante

Muito Importante

- ① Em C, cada instrução deve ser terminada com um ponto-e-vírgula (;), obtendo-se assim a linha 4 do programa -
`printf("Hello Mundo Cruel");`

Muito Importante

- ① Em C, cada instrução deve ser terminada com um ponto-e-vírgula (;), obtendo-se assim a linha 4 do programa -
`printf("Hello Mundo Cruel");`
- ② Preste muita atenção e perceba que o caractere " (aspas) é um **único** caractere e não pode ser substituído pelo caractere aspas simples ' nem muito menos por duas aspas simples ''.

Muito Importante

- ① Em C, cada instrução deve ser terminada com um ponto-e-vírgula (;), obtendo-se assim a linha 4 do programa -
`printf("Hello Mundo Cruel");`
- ② Preste muita atenção e perceba que o caractere " (aspas) é um **único** caractere e não pode ser substituído pelo caractere aspas simples ' nem muito menos por duas aspas simples ''.

Observação: A linguagem C não possui mecanismos de **Entrada/Saída (Input/Output)**. Dessa forma, precisamos recorrer a um conjunto de funções em **bibliotecas de funções (bibliotecas)**.

Programando em C

Dessa forma, precisamos adicionar à linguagem C um conjunto de funções que, “por defeito”, ela não nos proporciona.

Programando em C

Dessa forma, precisamos adicionar à linguagem C um conjunto de funções que, “por defeito”, ela não nos proporciona.

Para ter acesso a esse conjunto de funções teremos que incluir a sua definição no nosso código fonte.

Programando em C

Dessa forma, precisamos adicionar à linguagem C um conjunto de funções que, “por defeito”, ela não nos proporciona.

Para ter acesso a esse conjunto de funções teremos que incluir a sua definição no nosso código fonte.

No exemplo anterior fazemos isso com a linha `#include <stdio.h>`.

Programando em C

Programando em C

A linha #include <stdio.h> é C?

Resposta: A linha `#include <stdio.h>` **não** é C mas uma diretiva que indica ao pré-processador que deverá ser adicionado antes da compilação um arquivo existente em alguma parte no disco do computador. Tal arquivo nesse caso é **stdio.h**.

Programando em C

A linha #include <stdio.h> é C?

Resposta: A linha `#include <stdio.h>` **não** é C mas uma diretiva que indica ao pré-processador que deverá ser adicionado antes da compilação um arquivo existente em alguma parte no disco do computador. Tal arquivo nesse caso é **stdio.h**.

Esses arquivos são normalmente chamados de arquivos cabeçalhos e tem extensão .h, pois não têm código, mas apenas os cabeçalhos (**headers**) das funções que apresentam. Normalmente são chamados de **header files**.

Programando em C

A linha #include <stdio.h> é C?

Resposta: A linha `#include <stdio.h>` **não** é C mas uma diretiva que indica ao pré-processador que deverá ser adicionado antes da compilação um arquivo existente em alguma parte no disco do computador. Tal arquivo nesse caso é **stdio.h**.

Esses arquivos são normalmente chamados de arquivos cabeçalhos e tem extensão .h, pois não têm código, mas apenas os cabeçalhos (**headers**) das funções que apresentam. Normalmente são chamados de **header files**.

`#include <stdio.h>` = adiciona o arquivo **stdio.h** ao meu programa **exatamente nesta posição**.

Programando em C

**Por que não colocamos ; (ponto-e-vírgula) depois de
#include <stdio.h>?**

Resposta: Porque, como foi mencionado anteriormente,
`#include <stdio.h>` é uma diretiva do pré-processador. Dessa forma, `#include <stdio.h>` não é um comando de C.

Nota: Não colocamos ; (ponto-e-vírgula) ao final de instruções que não são de C.

O arquivo `stdio.h` permite o acesso a todas as funções de Entrada/Saída que necessitamos, em que **stdio** significa **standard input/output**.

Programando em C

O programa que imprime na tela a mensagem "Hello Mundo Cruel" poderia ser programado de uma forma diferente e equivalente da forma que segue.

Exemplo: Dividindo a mensagem em mais de um printf.

```
1: #include <stdio.h>
2: main()
3: {
4:     printf("Hello");
5:     printf(" ");
6:     printf("Mundo");
7:     printf(" ");
8:     printf("Cruel");
9: }
```

Programando em C

O compilador de C é, em geral, liberal no que diz respeito à forma como escrevemos (**identamos**) o código. Este pode ser escrito de modo que cada programador possa entender.

Programando em C

O compilador de C é, em geral, liberal no que diz respeito à forma como escrevemos (**identamos**) o código. Este pode ser escrito de modo que cada programador possa entender.

Exemplo: Compile o código C que segue abaixo.

```
#include <stdio.h>
main      ( )  {
    printf(
"Hello"
);
    printf(  "Mundo" )
;    printf(      "Cruel" )
; }
```

Programando em C

Observação: Espaços em branco são ignorados pelo compilador de C.

Nos programas que escrevem a frase “Hello Mundo Cruel” apresentados anteriormente observa-se que o cursor fica posicionado imediatamente após a palavra “Cruel”. Isso ocorre pelo fato de o programador não ter mandado o cursor mudar de linha após a escrita da mensagem.

Normalmente as linguagens de programação apresentam funções ou instruções distintas para realizar uma escrita na tela seguida ou não de uma nova linha. Em C a filosofia é outra.

Programando em C

Como podemos mudar de linha ao escrever uma mensagem?

Resposta: Tradicionalmente a mudança de linha é denominada

New Line, e em C é representada pelo símbolo \n.

Exemplo: Considere o exemplo que segue.

```
1: #include <stdio.h>
2: void main()
3: {
4:     printf("Hello Mundo Cruel\n");
5: }
```

Programando em C

Observação: O caractere especial *New Line* representado por \n é um caractere como qualquer outro. Assim, poderemos utilizar \n quantas vezes acharmos necessário. Nesse caso, poderíamos fazer:

Exemplo: Considere o exemplo que segue:

```
1: #include <stdio.h>
2: void main()
3: {
4:   printf("Hello\n\n Mundo\nCruel");
5: }
```

Programando em C

Exercício: Escreva um programa em C que possui a seguinte saída:

```
C  
eh uma otima linguagem. Eu serei  
um grande programador em  
C!
```

Programando em C

Exercício: Escreva um programa em C que possui a seguinte saída:

```
C  
eh uma otima linguagem. Eu serei  
um grande programador em  
C!
```

Uma solução:

```
1: #include <stdio.h>  
2: void main()  
3: {  
4:     printf("C\n");  
5:     printf("eh uma otima linguagem. Eu serei");  
6:     printf("um grande programado em");  
7:     printf("C!");  
8: }
```

Programando em C

Exercício:

Programando em C

Exercício: Outra solução utilizando apenas um printf()

```
#include <stdio.h>
void main()
{
    printf("C\neh uma otima linguagem.
    Eu seirei\um grande programador em\nC!");
}
```

Programando em C

Suponha agora que queremos escrever a seguinte mensagem na tela do computador.

Programando em C

Suponha agora que queremos escrever a seguinte mensagem na tela do computador.

Hoje eh um dia "LINDO" com uma aula linda!!!

Programando em C

Suponha agora que queremos escrever a seguinte mensagem na tela do computador.

Hoje eh um dia "LINDO" com uma aula linda!!!

Exemplo (com erro de compilação): Muitos seriam tentados a pensar no programa que segue.

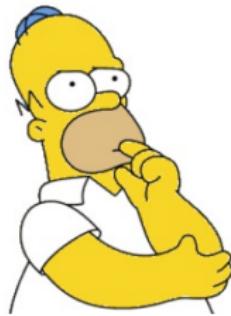
```
1: #include <stdio.h>
2: main()
3: {
4:     printf("Hoje eh um dia "LINDO" com uma aula
5:             linda!!!");
6: }
```

Por que do erro do código acima?

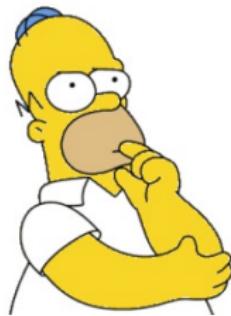
Resposta: O erro ocorre porque até a primeira ocorrência do símbolo " a linguagem C entende que estamos escrevendo "Hoje eh um dia ". Porém existe algo a mais dentro da função printf() que não é (*string*).

Nota: Chamamos de **string** uma cadeia de caracteres. Strings em C é definida entre aspas dupla (ex. "minha string").

Programando em C



Se as aspas servem para delimitar uma string, como poderemos escrever uma frase (string) em que as aspas sejam um caractere não delimitador?



Se as aspas servem para delimitar uma string, como poderemos escrever uma frase (string) em que as aspas sejam um caractere não delimitador?

Resposta: Coloque \ antes das aspas que se pretende escrever na string. Assim, as aspas serão tratadas como caracteres normais.

Programando em C

Exemplo: Para que possamos escrever a frase corretamente na tela do computador, considere o programa abaixo.

```
1: #include <stdio.h>
2: void main()
3: {
4:     printf("Hoje eh um dia \"LINDO\" com uma aula
5:             linda!!!");
6: }
```

Programando em C

Table: O caractere especial \.

\7	<i>Bell</i> (sinal sonoro do computador)
\a	<i>Bell</i> (sinal sonoro do computador)
\b	<i>BackSpace</i>
\n	<i>New Line</i> (mudança de linha)
\r	<i>Carroage Return</i>
\t	Tabulação Horizontal
\v	Tabulação Vertical
\\"	Caractere \
'	Caractere aspas simples '
"	Caractere aspas dupla "
?	Caractere ? (ponto de interrogação)
%%	Caractere %

Programando em C

Exercício: Crie alguns programas em C que faz uso de alguns caracteres especiais.

É muito importante a prática de se comentar muito bem um código em que escrevemos. Nunca se sabe quando teremos a necessidade de reprogramar o código.

É comum em um código não comentado esquecermos o que foi feito em um código fonte programado a muito tempo.

Observação: O bom programador segue “sempre” boas práticas de programação. Comentar bem um código por mais simples que ele seja é uma boa prática de programação.

Programando em C

Como são feitos comentários em C?

Resposta: Comentários em C são colocados entre /* */ (*/* AQUI É UM COMENTÁRIO */*).

Alguns exemplos de cabeçalhos de programas escritos em C são:

```
/* programa.c: Empresa Comentários e CIA */
/* AUTOR: Pedro Rafael */
/* DATA: 02/02/2017 */
```

Acima temos três comentários em que cada um deles estão entre /* */. Porém, poderíamos ter feito:

Programando em C

```
/* programa.c: Empresa Comentários e CIA
 * AUTOR: Pedro Rafael
 * DATA: 02/02/2017
 */
```

ou então:

```
*****  
* programa.c: Empresa Comentários e CIA  *  
* AUTOR: Pedro Rafael                      *  
* DATA: 02/02/2017                          *  
*****/
```

Observação: Os asteriscos horizontais ou verticais **não** tem nenhum significado especial. Eles apenas fazem parte do que queremos comentar.

Programando em C

Importante

Os comentários podem também serem colocados dentro de expressões ou instruções.

Exemplo: Poderíamos fazer:

```
printf("Bom dia querido professor, nós o adoramos!"  
/* É mentira :P */; /* Notar o ponto-ponto e vírgula  
*/
```

Dica: Particularmente, não acho uma boa prática de programação comentar no interior de instruções ou expressões. Tal postura tornará o código difícil de ser entendido.

Programando em C

Importante

Os compiladores de C **não** permitem, em geral, comentários dentro de comentários.

Exemplo: Comentários dentro de comentários não são permitidos, em geral, em C.

```
/* Meu primeiro comentário.  
/* Meu segundo comentário sobre o primeiro  
   comentário. */  
*/
```

Nota: Alguns compiladores permitem o uso de comentários dentro de comentários. Porém, tais opção não é reconhecida pelo ANSI.

Programação Aplicada à Estatística

Pedro Rafael Diniz Marinho

Universidade Federal da Paraíba

2021.2

Tipos de Dados em C



Sempre que abrimos nossa geladeira nos deparamos com uma enorme variedade de recipientes para cada tipo de produto: sólido, líquido, regulares, irregulares, etc.

Em C não é diferente. Precisamos conhecer aquilo que queremos armazenar e só então selecionar os melhores recipientes para a tarefa.

Tipos de Dados em C

Existem quatro tipos de dados básicos em C. São ele:

Tipos de Dados em C

Existem quatro tipos de dados básicos em C. São ele:

- ① **char;**

Tipos de Dados em C

Existem quatro tipos de dados básicos em C. São ele:

- ① **char;**
- ② **int;**

Tipos de Dados em C

Existem quatro tipos de dados básicos em C. São ele:

- ① **char;**
- ② **int;**
- ③ **float;**

Tipos de Dados em C

Existem quatro tipos de dados básicos em C. São eles:

- ① **char;**
- ② **int;**
- ③ **float;**
- ④ **double.**

A medida que formos adiantando o assunto entenderemos detalhadamente sobre cada um dos quatro tipos básicos de dados.

Observação: Existe ainda um outro tipo - tipo **Ponteiro** - que poderá ser considerado um tipo básico. Trataremos desse tipo quando estivermos um pouco mais adiantados no curso.

Variáveis

Sempre que temos o interesse de armazenar um valor que, por algum motivo, não seja fixo, devemos fazê-lo utilizando variáveis.

O que é um variável em C?

Resposta: Uma **variável** é nada mais que um nome que damos a uma determinada posição na memória para conter um valor de um determinado tipo.

Como o próprio nome indica, o valor contido em uma variável pode variar ao longo da execução de um programa.

Muito Importante

Uma variável deve ser **sempre** definida **antes** de ser utilizada. O compilador normalmente indica a não declaração de uma variável na mensagem de aviso contendo o erro de compilação.

Na definição de uma variável, precisamos especificar ao compilador qual o tipo de dado que fica atribuído ao nome que indicamos para essa variável.

Variáveis

Declarando uma Variável

A definição de variáveis é feita utilizando a seguinte sintaxe:

```
tipo var1 [, var2, ..., varn] ;
```

Observação: Tudo que está dentro de colchetes é opcional e os colchetes não faz parte da declaração. Por exemplo, se quisermos declarar duas variáveis do mesmo tipo, podemos fazer: tipo var₁, var₂;. O mesmo raciocínio deverá ser seguido para três ou mais variáveis.

Variáveis

Exemplo: O trecho de código abaixo ilustra a declaração dos tipos básicos de variáveis em C.

```
int i;      /* i é uma variável do tipo inteiro */
char ch1, novo_char; /* ch1 e novo_char é uma variável
do tipo char */
float pi, raio, perimetro; /* Estas são variáveis do
tipo float */
double total, k123; /* Aqui estão as variáveis do
tipo double */
```

Variáveis

Sim, eu sou chato 😊 e vou repetir novamente: **Uma variável deverá ser declarada antes de sua utilização e antes de qualquer instrução. O esquema abaixo ilustra isso:**

```
main()
{
    Declaração de variáveis;

    Instrução 1;
    Instrução 2;
}
```

Variáveis

As variáveis são armazenadas em um endereço de memória do computador. Por meio do nome da variável é que acessamos a posição na memória que armazena o tipo de dado básico que armazenamos.

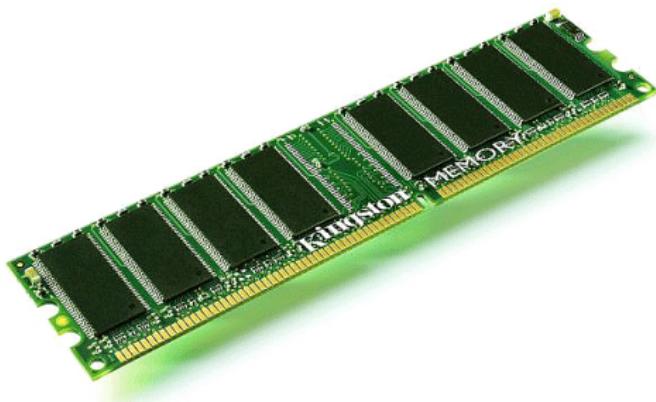


Figura: Memória de um computador digital.

Variáveis

É muito mais conveniente e simples referenciar os endereços da memória por meio de nomes de variáveis. O tipo que está associado ao nome da variável indica a quantidade de *bytes* que serão utilizados para guardar um valor nessa variável.

Para nomear as variáveis, é conveniente observar um conjunto de regras que serão listadas abaixo:

Variáveis

É muito mais conveniente e simples referenciar os endereços da memória por meio de nomes de variáveis. O tipo que está associado ao nome da variável indica a quantidade de *bytes* que serão utilizados para guardar um valor nessa variável.

Para nomear as variáveis, é conveniente observar um conjunto de regras que serão listadas abaixo:

- O nome de uma variável pode ser construído por letras do alfabeto (minúsculas ou maiúsculas), dígitos (de 0 a 9) e ainda pelo caractere underscore (_).

Variáveis

É muito mais conveniente e simples referenciar os endereços da memória por meio de nomes de variáveis. O tipo que está associado ao nome da variável indica a quantidade de *bytes* que serão utilizados para guardar um valor nessa variável.

Para nomear as variáveis, é conveniente observar um conjunto de regras que serão listadas abaixo:

- O nome de uma variável pode ser construído por letras do alfabeto (minúsculas ou maiúsculas), dígitos (de 0 a 9) e ainda pelo caractere underscore (_).
- O primeiro caractere **não** poderá ser um dígito. Terá que ser uma letra ou o caractere underscore.

Variáveis

Variáveis

- O uso do caractere underscore no início de uma variável é desaconselhável mas não é errado.

Variáveis

- O uso do caractere underscore no início de uma variável é desaconselhável mas não é errado.
- Uma variável jamais poderá ter o nome de uma palavra reservada em C como main, int, float, char, double, if, while, else, return, goto, entre diversas outras.

Variáveis

- O uso do caractere underscore no início de uma variável é desaconselhável mas não é errado.
- Uma variável jamais poderá ter o nome de uma palavra reservada em C como main, int, float, char, double, if, while, else, return, goto, entre diversas outras.
- Lembre-se que a linguagem C é *Case Sensitive*. Assim, int a é diferente de declarar int A.

Variáveis

- O uso do caractere underscore no início de uma variável é desaconselhável mas não é errado.
- Uma variável jamais poderá ter o nome de uma palavra reservada em C como main, int, float, char, double, if, while, else, return, goto, entre diversas outras.
- Lembre-se que a linguagem C é *Case Sensitive*. Assim, int a é diferente de declarar int A.

Observação: Evite a declaração de variáveis utilizando caracteres acentuados (ã, á, à, ó, etc.). A maioria dos compiladores não aceita tais caracteres como admissíveis.

Variáveis

Exemplo: Declarações de variáveis em C.

```
int idade; /* Correto */  
int Num_Cliente; /* Correto */  
float a1b2c3; /* Correto */  
float 7a2b3c; /* Incorreto */  
char float; /* Incorreto */  
double vinte%; /* Incorreto */  
char sim?não; /* Incorreto */  
int _alfa; /* Correto */  
int _123; /* Correto */  
char num, NUM; /* Correto */
```

Dica: Na segunda declaração vemos um bom uso do caractere underscore que foi utilizado para tornar mais legível a variável.

Variáveis

Quantos caracteres podemos utilizar para definir uma variável?

Resposta: O número de caracteres irá depender do compilador de C utilizado. Normalmente é permitido nomes de variáveis com até 32 caracteres (ou mais).

Algumas boas práticas de programação são:

- O nome de uma variável deve nos lembrar do que ela de fato armazena;
- O nome de uma variável não deve ser todo escrito em maiúscula. Normalmente a maioria dos programadores definem constantes dessa forma.
- Caso o nome de uma variável utilize mais de uma palavra, utilize a dica do *frame* anterior.

Variáveis

Nota: Quando uma variável é declarada fica sempre com um valor, qual resulta do estado aleatório de bits que a constituem. Isto é, uma variável declarada sempre possui um lixo no seu conteúdo.

Porém, é sempre conveniente inicializar uma variável com um valor através de uma operação de atribuição. Assim, ao realizar uma atribuição, o valor antigo na memória ao qual a variável ocupa é eliminado, ficando nela o novo valor atribuído.

A operação de atribuição obedece a seguinte sintaxe:

variável = expressão ou valor atribuído;

Variáveis

A atribuição de valores em C é realizada através do sinal = e a variável **SEMPRE** é colocada ao **lado esquerdo** da atribuição.

Variáveis

A atribuição de valores em C é realizada através do sinal = e a variável **SEMPRE** é colocada ao **lado esquerdo** da atribuição.

Exemplo:

```
int num; /* Declaração da variável num */  
num = -17; /* num recebe o valor -17 */
```

Variáveis

A atribuição de valores em C é realizada através do sinal = e a variável **SEMPRE** é colocada ao **lado esquerdo** da atribuição.

Exemplo:

```
int num; /* Declaração da variável num */  
num = -17; /* num recebe o valor -17 */
```

Observação: Podemos fazer uma atribuição à variável no momento de sua declaração.

Variáveis

A atribuição de valores em C é realizada através do sinal = e a variável **SEMPRE** é colocada ao **lado esquerdo** da atribuição.

Exemplo:

```
int num; /* Declaração da variável num */  
num = -17; /* num recebe o valor -17 */
```

Observação: Podemos fazer uma atribuição à variável no momento de sua declaração.

Exemplo:

```
int num = -17;  
int n1 = 3, n2 = 5;  
int a = 10, b, c = -123, d;
```

Variáveis

Observação: Podemos passar o valor contido no endereço de memória de uma variável para o endereço de memória de outra variável.

Exemplo:

```
int a = 3, b = 7;  
a = b; /* a variável "a" conterá o valor 7 */
```

Variáveis

Nota: Em C é possível atribuir o mesmo valor a várias variáveis.

Exemplo: Colocar o valor 5 nas variáveis a, b, c e d previamente declaradas:

a = 5;

b = 5;

c = 5

d = 5;

Seria mais conveniente fazer a = b = c = d = 5;. Tal declaração está correta.

Observação: As atribuições em C são realizadas da direita para a esquerda.

Inteiros - int

As variáveis do tipo **int** são utilizadas para armazenar valores naturais (sem parte fracionária) positivos e negativos. Por exemplo, 2, 0, 7, -345, +115.

Table: Operações sobre números inteiros.

Operação	Descrição	Exemplo	Resultado
+	Soma	$21 + 4$	25
-	Subtração	$21 - 4$	17
*	Multiplicação	$21 * 4$	84
/	Divisão	$21 / 4$	5
%	Módulo	$21 \% 4$	1

Inteiros - int

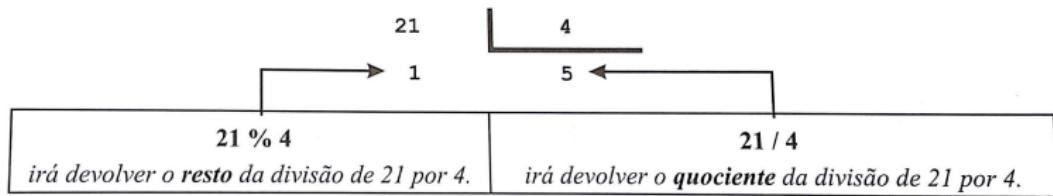
Observação: O **módulo** trata-se do resto da divisão inteira. O operador / fornece o quociente da divisão de dois inteiros. Já o operador % (módulo) fornecerá o resto da divisão de dois números inteiros.

Muito Importante

Qualquer operação entre inteiros retorna sempre um inteiro. Nesse caso, por exemplo, $21/4$ retornará não irá retornar o valor 5.25 como poderíamos pensar. Teremos nesse caso o retorno do valor 5.

Inteiros - int

A imagem abaixo apresenta bem a diferença do uso dos operadores / e %.



Inteiros - int

Consideremos com atenção o código no exemplo que segue:

Exemplo:

```
1: #include <stdio.h> /* Essa linha não é código C */
2:
3: void main()
4: {
5:     int num = 123;
6:
7:     printf("O valor de num = %d e o
8:             valor seguinte = %d\n", num, num+1);
9: }
```

O que o programa imprime em tela?

Inteiros - int

Observação: Sempre que estamos interessados em escrever o valor de um inteiro dentro de um **printf** devemos substituir o valor desse inteiro por um **formato de escrita**.

Em geral, qual o formato de escrita de um inteiro em C?

Resposta: O formato de escrita de um inteiro na função **printf()** é **%d**.

Devemos colocar **%d** na função **printf()** no lugar onde queremos substituir o valor armazenado na variável inteira. Ao final da string, especificamos de quais variáveis esses valores serão obtidos.

Inteiros - int

Da mesma forma que existe uma função `printf()` para escrita de valores, existe uma função correspondente para a leitura de valores, está é a função `scanf()`.

Exercício: Implemente o código abaixo e explique cada linha de código do programa que segue.

```
1: #include <stdio.h>
2:
3: void main()
4: {
5:     int num;
6:     printf("Introduza um numero: ");
7:     scanf("%d", &num);
8:     printf("O numero introduzido foi %d\n", num);
9: }
```

Função scanf

A função `scanf` = `scan` + `formatado` funciona de forma semelhante à função `printf`. Uma vez que ela foi implementada para a leitura de valores, a *string* inicial (primeiro argumento da função) deve conter apenas o formato das variáveis que queremos ler. **Exceto se a variável for uma string.**

Depois de especificados os formatos de leitura na *string*, devem ser colocadas todas as variáveis correspondentes pela ordem em que ocorrem os formatos, precedidas de um &.

Observação: O esquecimento do (“E” comercial) poderá gerar resultados inesperados. Preste atenção no seu uso!

Inteiros - int

Observação: Na função `scanf` preste atenção que o tipo de leitura “%d” necessita que a variável `num` tenha o mesmo tipo que nesse caso é inteiro.

Importante

A string enviada para a função `scanf` não deve conter outros caracteres que não sejam os caracteres indicadores de formato. Jamais coloque `\n` na *string* da função `scanf`.

Inteiros e Variações

Os tipos de dados variam de arquitetura para arquitetura. No caso do tipo **int** são habituais os valores de 2 ou 4 *bytes*. Sendo assim, é importante saber qual a dimensão de um inteiro quando se desenvolve uma aplicação, caso contrário corremos o risco de tentar armazenar um valor em uma variável inteira com o número de *bytes* insuficiente.

E como eu sei a dimensão de um int no meu computador?

Resposta: A linguagem C disponibiliza um operador denominado `sizeof`. Seu uso é da forma que segue:

```
sizeof <expressão> ou sizeof (<tipo de dados>)
```

Nota: <> não faz parte da sintaxe!

Inteiros e Variações

Exemplo: O programa abaixo indica o número de **bytes** que ocupa um inteiro.

```
1: #include <stdio.h>
2:
3: main()
4: {
5:     printf("O tamanho em bytes de um inteiro = %d\n",
6:            sizeof(int));
7: }
```

Exercício: Crie um programa que apresente tamanho em número de *bytes* dos tipos de dados **int**, **char**, **float** e **double**.

Inteiros e Variações

Importante

O fato de o tamanho de um inteiro poder variar de arquitetura para arquitetura é algo preocupante, pois os limites das variáveis que armazenam um inteiro podem variar de maneira drástica. Isso reduz consideravelmente a portabilidade dos programas entre máquinas diferentes. Veja a tabela abaixo:

Tabela: Variação de inteiros pela quantidade de *bytes*.

Número de Bytes	Menor Valor	Maior Valor
2	-32768	32767
4	-2147483648	2147483647

Inteiros e Variações

A forma que podemos garantir que o programa venha sempre utilizar a mesma quantidade de *bytes* independente da arquitetura é utilizando quatro prefixos distintos para melhor definição da característica da variável. São eles:

- ① **short** - Inteiro pequeno (2 *bytes*);
- ② **long** - Inteiro grande (4 *bytes*);
- ③ **signed** - Inteiro com sinal (números negativos e positivos);
- ④ **unsigned** - Inteiro sem sinal (apenas números positivos).

Inteiros e Variações

Para que possamos garantir que o inteiro n utiliza apenas 2 *bytes* de memória, independentemente da arquitetura utilizada, devemos declarar a variável como

```
short int n; /* ou short n; */
```

Já para que possamos garantir de o inteiro n usa sempre 4 *bytes* de memória, independentemente da arquitetura utilizada, devemos declarar a variável como

```
long int n; /* ou long n; */
```

Observação: O prefixo **short** garante o tamanho mínimo do inteiro e o prefixo **long** garante o tamanho máximo.

Inteiros e Variações

Tabela: Quantidade de *bytes* reservadas em diferentes tipos de inteiros.

Sistema (bits)	short int	int	long int
32 bits	2	2	4
64 bits	2	4	8

Importante

O formato de leitura e escrita de variáveis inteiras short e long nas funções scanf e printf deve ser precedido dos prefixos **h** (short) e **l** (long).

Inteiros e Variações

Exercício: Compile o programa e explique cada linha de código.

```
1: #include <stdio.h>
2:
3: void main()
4: {
5:     short int idade;      /* ou short idade */
6:     int montante;
7:     long int n_conta;    /* ou long n_conta */
8:
9:     printf("Qual a Idade? ");
10    scanf("%hd", &idade);
11    printf("Qual o montante a depositar? ");
12    scanf("%d", &montante);
13    printf("Qual o numero da conta? ");
```

Inteiros e Variações

```
14:     scanf("%ld", &n_conta);
15:     printf("Uma pessoa de %hd anos depositou R$%d reais
16                 na conta %ld\n", idade, montante, n_conta);
```

Prefixos `signed` e `unsigned`

Uma variável do tipo inteiro admite valores inteiros **positivos** e **negativos**. Caso se deseje que a variável contenha apenas valores positivos, devemos declarar com o prefixo **`unsigned`**.

Uma pequena sequência de exercícios

Exercício: Crie um programa em C que solicite ao usuário para digitar 2 valores inteiros e coloque o primeiro valor digitado em uma variável **a** e o segundo valor digitado em uma variável **b**. Em seguida, o programa deverá imprimir em tela o conteúdo dessas variáveis.

Uma pequena sequência de exercícios

Exercício: Crie um programa em C que solicite ao usuário para digitar 2 valores inteiros e coloque o primeiro valor digitado em uma variável **a** e o segundo valor digitado em uma variável **b**. Em seguida, o programa deverá imprimir em tela o conteúdo dessas variáveis.

Exercício: Cria um programa em C que solicite ao usuário para digitar 4 valores inteiros e armazene os valores nas variáveis **a**, **b**, **c** e **d**, respectivamente. O seu programa deverá retornar o resultado da seguinte expressão matemática:

$$(a + b + c) \times d.$$

Uma pequena sequência de exercícios

Exercício: Construa um programa que realize uma conversão de valores de reais para dólares. Dessa forma, o usuário deverá ser solicitado do valor em reais e deverá imprimir em tela o valor equivalente em dólares americano. Considere que R\$ 1.00 reais equivale à \$ 0.30796 dólares. (**Dica:** Como nesse ponto sabemos definir variáveis reais, não há a necessidade de considerar apenas valores inteiros.)

Uma pequena sequência de exercícios

Exercício: Construa um programa que realize uma conversão de valores de reais para dólares. Dessa forma, o usuário deverá ser solicitado do valor em reais e deverá imprimir em tela o valor equivalente em dólares americano. Considere que R\$ 1.00 reais equivale à \$ 0.30796 dólares. (**Dica:** Como nesse ponto sabemos definir variáveis reais, não há a necessidade de considerar apenas valores inteiros.)

Exercício: A condição física de uma pessoa pode ser medida com base no cálculo do Índice de Massa Corporal (IMC). O mesmo é calculado dividindo-se o peso desta pessoa em kg pelo quadrado sua altura em m. Escreva um programa que leia o peso em kg e a altura de uma pessoa em m, calcule e mostre o IMC. Se as entradas fossem 70.0kg para o peso 1.80m para altura então a saída seria aproximadamente 21.60.

Uma pequena sequência de exercícios

Exercício: Leia uma temperatura em graus Celsius e apresente-a convertida em graus Fahrenheit. Considere a seguinte fórmula de conversão:

Uma pequena sequência de exercícios

Exercício: Leia uma temperatura em graus Celsius e apresente-a convertida em graus Fahrenheit. Considere a seguinte fórmula de conversão:

$$F = C * (9.0 / 5.0) + 32.0,$$

em que F é a temperatura em graus Fahrenheit e C é a temperatura fornecida pelo usuário do programa em graus Celsius.

Uma pequena sequência de exercícios

Exercício: Leia uma temperatura em graus Celsius e apresente-a convertida em graus Fahrenheit. Considere a seguinte fórmula de conversão:

$$F = C * (9.0 / 5.0) + 32.0,$$

em que F é a temperatura em graus Fahrenheit e C é a temperatura fornecida pelo usuário do programa em graus Celsius.

Exercício: Faça o mesmo programa porém que converta a temperatura de graus Celsius para graus Fahrenheit.

Uma pequena sequência de exercícios

Exercício: Construa um programa que leia um ângulo em graus e apresente-o convertido em radianos. Lembre-se:

Uma pequena sequência de exercícios

Exercício: Construa um programa que leia um ângulo em graus e apresente-o convertido em radianos. Lembre-se:

$$R = G * \frac{\pi}{180},$$

em que G é o ângulo informado pelo usuário em graus e R o valor da conversão em radianos. (**Dica:** Considere $\pi = 3.1415926535$.)

Exercício: Implemente um programa em C para o dono de uma empresa. O mesmo está interessado em um programa em que ao se inserir um salário base, o programa imprima o valor recebido, a gratificação de 5% calculada em cima do salário base, o imposto a pagar que deverá ser inserido pelo usuário e calculado também em cima do salário base e o valor a receber após gratificações e deduções.

Uma pequena sequência de exercícios

Exercício: Construa um programa que possa ser utilizado por um construtor. O mesmo deseja um programa simples em que possa ser inserido o preço do metro quadrado de um terreno de uma dada região geográfica. O construtor deseja inserir as medidas em metros de largura e comprimento do terreno. O programa deverá imprimir as medidas inseridas, calcular o metro quadrado e o valor do terreno com base no valor do metro quadrado inserido pelo construtor (usuário do programa).

Uma pequena sequência de exercícios

Exercício: Construa um programa que possa ser utilizado por um construtor. O mesmo deseja um programa simples em que possa ser inserido o preço do metro quadrado de um terreno de uma dada região geográfica. O construtor deseja inserir as medidas em metros de largura e comprimento do terreno. O programa deverá imprimir as medidas inseridas, calcular o metro quadrado e o valor do terreno com base no valor do metro quadrado inserido pelo construtor (usuário do programa).

Exercício: Três amigos jogaram na loteria. Caso eles ganhem, o prêmio deve ser repartido proporcionalmente ao valor investido que cada um deu para a realização da aposta coletiva. Construa um programa em C que leia e imprima o valor apostado por cada um, o percentual do valor investido por cada um e quanto cada um irá ganhar no repartimento do valor total do prêmio.

Inteiros e Variações

Exemplo:

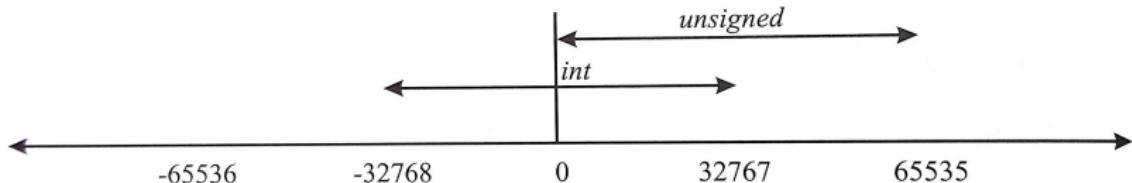
```
unsigned int Idade;      /* ou unsigned Idade; */  
/* A Idade de um indivíduo  
não pode ser negativa */
```

Importante

O prefixo **signed** antes de um inteiro não é necessário, pois por padrão todos os inteiros que são criados são sinalizados (**signed**).

Inteiros e Variações

Ao trabalhar com valores sem sinal, o conjunto de valores com que podemos trabalhar do lado positivo é ampliado uma vez que não utilizamos *bytes* para especificar o sinal do inteiro. Observe sua variação abaixo:



Importante e nunca esqueça 😊

O formato de leitura e escrita de variáveis sem sinal (**unsigned int**), utilizando as funções `scanf` e `printf` é `%u` em vez de `%d`.

Supondo que o tamanho de um inteiro é de 2 *bytes*, apresenta-se em seguida a lista de limites de valores em que uma variável do tipo inteiro pode variar dependendo do prefixo utilizado.

Inteiros e Variações

Tabela: Representação de valores de uma variável do tipo `int` com diferentes tipos de prefixos.

Tipo de Variável	Nº de Bytes	Valor Mínimo	Valor Máximo
int	2	-32 768	32 767
short int	2	-32 768	32 767
long int	4	-2 147 483 648	2 147 483 647
unsigned int	2	0	65 535
unsigned short int	2	0	65 535
unsigned long int	4	0	4 294 967 295

Observação: Note que especificar **int** é o mesmo que fazer **short int**. Já especificar **unsigned short int** equivale à utilizar **unsigned int**.

Float e Double

As variáveis declaradas do tipo **float** ou **double** são utilizadas para armazenar valores numéricos com parte fracionária. São também frequentemente denominadas reais ou de ponto flutuante.

Exemplo: 3.14, 0.0007635, 1.0.

Qual a diferença de float para double?

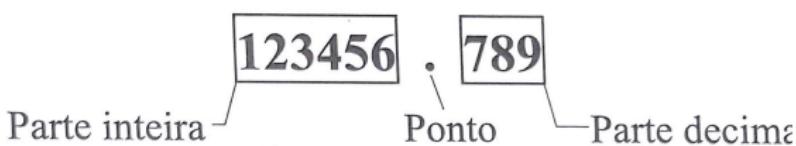
Resposta: A diferença entre o tipo **float** e **double** consiste no número de *bytes* que é reservado na memória para armazenar o valor. A dimensão do **float** é normalmente de quatro *bytes*, enquanto a do **double** é de oito *bytes*.

Float e Double

Observação: É comum dizer que esses dois tipos armazenam números com precisão simples (**float**) ou com precisão dupla (**double**).

Importante

Uma variável real é representada por uma parte inteira e por outra decimal que são separadas por um ponto. Diferentemente do que ocorre no nosso idioma, não utilizamos vírgula para separar essas partes.



Float e Double

Exemplo: Definindo variáveis reais em C.

```
1: float Pi = 3.1415; /* Precisão Simples */  
2: double erro = 0.000001 /* Precisão Dupla */  
3: float total = 0.0; /* 0.0 é diferente de 0 para o  
4:                      computador */
```

Float e Double

Exercício: Escreva um programa em C que calcule o perímetro e a área de uma circunferência.

Float e Double

Exercício: Escreva um programa em C que calcule o perímetro e a área de uma circunferência.

```
1: #include <stdio.h>
2:
3: void main()
4: {
5:     float raio, perimetro;
6:     double Pi = 3.1415927, area;
7:
8:     printf("Introduza o Raio da Circunferência: ");
9:     scanf("%f", &raio);
10:    area = Pi * raio * raio;
11:    perimetro = 2 * Pi * raio;
```

Float e Double

```
12:  
13:     printf("Area = %f\nPerimetro = %f\n",
14:                 area, perimetro);  
15: }
```

Qual o formato de leitura e escrita para números reais?

Resposta: O formato de leitura e escrita para números reais é `%f`.

Importante

Em C não existe qualquer operador que permita calcular o quadrado de um número. Este terá que ser calculado através da multiplicação desse número por si próprio ou através da função `pow` da biblioteca `math.h`.

Float e Double

É possível utilizar notação científica?

Resposta: Sim. Isso poderá ser feito da forma que segue:

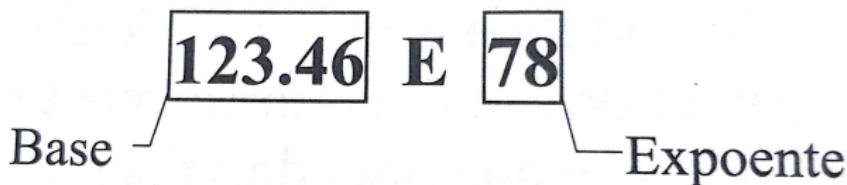


Figura: Formato de um número em notação científica.

Observação: O número armazenado na imagem acima é o valor $123.46 \cdot 10^{78}$.

Float e Double

Importante

É importante notar que a base tem que ser real. Sendo assim, para representar o valor 1000 em notação científica, em C, é preciso fazer 1.0E3 ou 1.0E+3. De forma análoga poderemos fazer 1.0E-3 ou 1.0e-3 para representar o valor 0.001.

Observação: Podemos, em C, substituir E por e, isto é, temos que 1.0E3 é equivalente à fazer 1.0e3.

Exemplo: Escreva um programa que realize a conversão de toneladas para quilos e gramas escrevendo o resultado em notação tradicional ($aaaa.bbb$) e científica ($aaaE \pm bb$).

Float e Double

```
1: #include <stdio.h>
2:
3: void main()
4: {
5:     float quilos = 1.0E3; /* Uma tonelada são
6:                           1000 quilos */
7:     double gramas = 1.0e6; /* Uma tonelada são
8:                           1 000 000 gramas */
9:     float n_toneladas;
10:
11:    printf("Quantas toneladas comprou: ");
12:    scanf("%f", &numero_toneladas);
13:    printf("N. de Quilos = %f = %e\n",
14:           n_toneladas * quilos,
15:           n_toneladas * quilos);
```

Float e Double

```
16:     printf("N. de gramas = %f = %E\n",
17:                 n_toneladas * gramas,
18:                 n_toneladas * gramas);
19: }
```

Importante

O formato de leitura e escrita de números reais em notação científica é dado por %e ou %E. Porém, caso o número seja lido com %f (com scanf) e imprimido em tela com printf não há problemas. O contrário também é verdadeiro, isto é, podemos entrar com um número com o formato %e ou %E (usando scanf) e podemos imprimir seu resultado com a função printf utilizando o formato %f.

Importante

Qualquer operação que inclua um dos operandos do tipo real obtém um resultado do tipo real. A diferença entre um inteiro e um número real está na presença ou na ausência de um **ponto**, que é o separador das partes inteiras e fracionária.

Importante

Qualquer operação que inclua um dos operandos do tipo real obtém um resultado do tipo real. A diferença entre um inteiro e um número real está na presença ou na ausência de um **ponto**, que é o separador das partes inteiras e fracionária.

- 10 em C é um **número inteiro**;

Importante

Qualquer operação que inclua um dos operandos do tipo real obtém um resultado do tipo real. A diferença entre um inteiro e um número real está na presença ou na ausência de um **ponto**, que é o separador das partes inteiras e fracionária.

- 10 em C é um **número inteiro**;
- -10 em C é um **número inteiro**;

Operações sobre Reais

Importante

Qualquer operação que inclua um dos operandos do tipo real obtém um resultado do tipo real. A diferença entre um inteiro e um número real está na presença ou na ausência de um **ponto**, que é o separador das partes inteiras e fracionária.

- 10 em C é um **número inteiro**;
- -10 em C é um **número inteiro**;
- 10.0 em C é um **número real**;

Operações sobre Reais

Importante

Qualquer operação que inclua um dos operandos do tipo real obtém um resultado do tipo real. A diferença entre um inteiro e um número real está na presença ou na ausência de um **ponto**, que é o separador das partes inteiras e fracionária.

- 10 em C é um **número inteiro**;
- -10 em C é um **número inteiro**;
- 10.0 em C é um **número real**;
- 10. em C é um **número real**;

Operações sobre Reais

Importante

Qualquer operação que inclua um dos operandos do tipo real obtém um resultado do tipo real. A diferença entre um inteiro e um número real está na presença ou na ausência de um **ponto**, que é o separador das partes inteiras e fracionária.

- 10 em C é um **número inteiro**;
- -10 em C é um **número inteiro**;
- 10.0 em C é um **número real**;
- 10. em C é um **número real**;
- 10.25 em C é um **número real**.

Operações sobre Reais

Nunca Esqueça

Qualquer operação em que pelo menos um dos operandos seja real produz um resultado do tipo real. Se algum dos operandos for, por exemplo, inteiro e um outro real, o inteiro é alterado para o tipo real para que se possa realizar a operação entre dois reais.

Exemplo:

```
21 / 4 /* = 5; divisão inteira */  
21.0 / 4 /* = 5.25; um dos operandos é real */  
21 / 4. /* = 5.25; um dos operandos é real */  
21.0 / 4.0 /* = 5.25; ambos operandos são reais */
```

Operações sobre Reais

Posso fazer qualquer operação entre números reais em C?

Resposta: Quase todas. A única operação que não podemos utilizar entre números reais é a operação % (módulo - resto da divisão). A operação módulo da divisão poderá apenas ser aplicado se ambos os operandos forem inteiros.

Salvo os erros de arredondamento inerentes à qualquer processador, teoricamente o resto da divisão entre dois números reais sempre será zero. Dessa forma, não faz sentido procurar o resto da divisão de dois números reais por meio de uma função C.

Caracteres - char

O tipo caractere permite armazenar **UM ÚNICO CARACTERE** à uma variável do tipo **char**.

Nota: Um dos erros mais comum na programação em C é pensar que uma variável declarada com o tipo **char** armazena mais de um caractere, isto é, pensar que estas variáveis armazenam *strings*.

Como um *byte* possui 8 *bits*, temos que uma variável do tipo **char** poderá armazenar 256 ($2^8 = 256$, arranjo com reposição) caracteres possíveis.

00000000 - Todos os bits com 0 (valor 0)

11111111 - Todos os bits com 1 (valor 255)

Números Binários

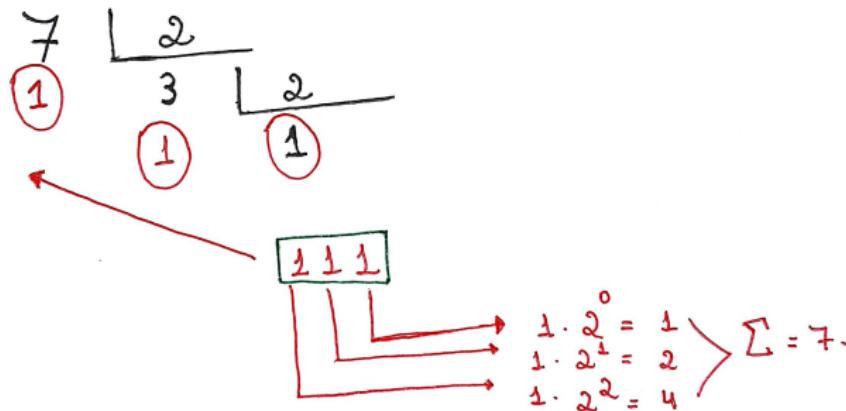
Como converter números binários para inteiro e números inteiros para binário?

Números Binários

Exemplo: Conversão do número inteiro 7 para números binários.
Conversão do número binário 111 para o número inteiro.

Números Binários

Exemplo: Conversão do número inteiro 7 para números binários.
Conversão do número binário 111 para o número inteiro.



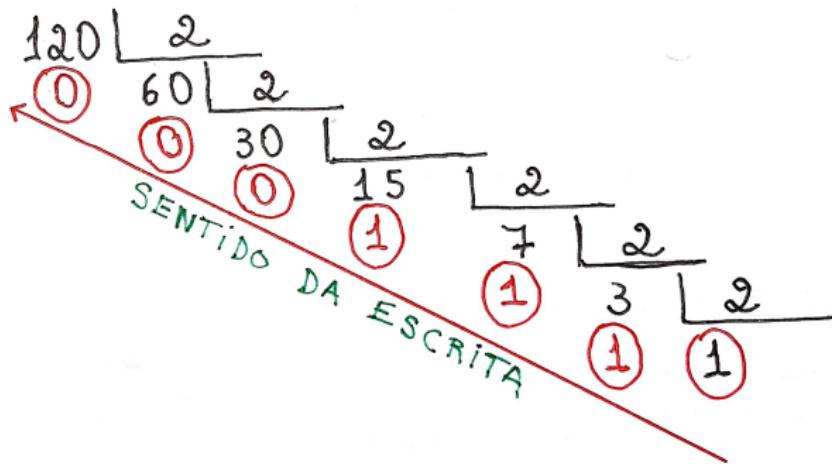
Observação: Se o computador for de 32 bits, ele enxergará o número 7 como: 00000 00000 00000 00000 00000111. Dessa forma, basta escrever o número binário obtido e acrescentar a quantidade de zeros necessárias antes do número até se obter 32 bits.

Números Binários

Exemplo: Convertendo o número 120 para binário.

Números Binários

Exemplo: Convertendo o número 120 para binário.



Neste caso, a escrita do número 120 em binário é 1111000. Em um computador de 32 bits o número 120 é compreendido como 00000 00000 00000 00000 1111000.

Caracteres - char

Sendo assim, como um caractere tem apenas um *byte*, isto é, possui apenas 8 *bits*, temos que o último caractere representado é o 255. Para se convencer disso, faça o exercício que segue.

Exercício: Converta o número binário 11111111 para número inteiro.

Dessa forma, poderemos representar caracteres de 0 a 255, isto é, são 256 caracteres representáveis. Cada um dos 256 caracteres estará associado a um número inteiro de 0 a 255.

Importante: O formato de leitura e escrita de uma variável do tipo **char** é `%c`.

Caracteres - char

Importante

Jamais esqueça que a representação de um caractere em C faz-se utilizando aspas simples ('A') e **não** ("A"). Aspas duplas utilizaremos para representar uma cadeia de caracteres (*string*).

Exemplo: Exemplo de inicialização de variáveis do tipo **char**.

```
char ch = 'A', v = '$', n1 = '\n'
```

Observação: Utilizar caracteres e passar como argumentos para variáveis do tipo **char** é algo fácil uma vez que a maioria dos caracteres que necessitamos estão dispostos em nosso teclado. Porém, é possível utilizar outros caracteres pertencente à tabela ASCII. Veja o link com a tabela completa:

<https://goo.gl/couXal>.

Exercício: Selecione 20 números com representação na base 10 da tabela <https://goo.gl/couXal> e converta-os para binário. Faça o processo inverso, converta-os de binário para decimal.

Caracteres - char

Observação: Lembre-se dos caracteres especiais em C apresentados anteriormente que só ocupam um único *byte*. Dessa forma, é possível passar como argumento à uma variável do tipo **char** um caractere especial.

Exemplo:

```
1: #include <stdio.h>
2:
3: void main()
4: {
5:     char meu_caractere = '\n';
6:     printf(Meu caractere eh: %c%c%c, meu_caractere,
7:             meu_caractere, meu_caractere);
8: }
```

Caracteres - char

Exemplo: O programa seguinte é uma outra forma de escrever a string **Hello Mundo Cruel**.

```
1: #include <stdio.h>
2:
3: void main()
4: {
5:     printf("%cello %cund%c %c%cue%c",
6:            'H', 'M', 'o', 'C', 'r', 'l');
7: }
```

Caracteres - char

Exercício: Escreva um programa que leia um caractere enviado pelo teclado e em seguida o escreva na tela.

Caracteres - char

A função getchar()

A leitura de caracteres pode ser realizada sem ter que se recorrer à função scanf. Tal função é desenhada **unicamente** para a leitura de caracteres e não possui nenhum parâmetro.

Exemplo:

```
1: #include <stdio.h>
2:
3: void main()
4: {
5:     char ch;
6:     printf("Introduza um Caractere: ");
7:     ch = getchar();
8:     printf("O caractere introduzido foi %c\n", ch);
```

Caracteres - char

Exercício: Escreva um programa em C que solicite através da função scanf um caractere ao usuário e logo em seguida solicite outro. Depois de introduzido ambos os caracteres, tal programa deverá mostrar os dois caracteres lidos entre aspas simples.

Caracteres - char

```
1: #include <stdio.h>
2:
3: void main()
4: {    char ch1, ch2;
5:     printf("Introduza um Caractere: ");
6:     scanf("%c", &ch1);
7:     printf("Introduza um outro Caractere: ");
8:     scanf("%c", &ch2);
9:     printf("Os caracteres introduzidos foram \'%c\' e
10:           \'%c\'\n", ch1, ch2);
11:}
```

Caracteres - char

Por que do resultado estranho do programa anterior?

Caracteres - char

Por que do resultado estranho do programa anterior?

Resposta: Isso se deve ao fato do ENTER também ser um caractere definido na tabela ASCII. Além disso, o teclado do nosso computador possui uma memória chamada de memória *buffer* que é um repositório temporário de caracteres que nós escrevemos.

Caracteres - char

Por que o resultado é estranho do programa anterior?

Resposta: Isso se deve ao fato do ENTER também ser um caractere definido na tabela ASCII. Além disso, o teclado do nosso computador possui uma memória chamada de memória *buffer* que é um repositório temporário de caracteres que nós escrevemos.

Enquanto não pressionamos a tecla **ENTER**, o conteúdo do *buffer* não é enviado às variáveis. Ao pressionar **ENTER**, o próximo conteúdo do *buffer* é o caractere **ENTER**. Assim, estamos passando dois caracteres ao invés de um. Dessa forma, o **ENTER** é passado ao segundo scanf.

Caracteres - char

Observação: Esse “problema” ocorre tanto com a função `scanf` como com a função `getchar`!

Caracteres - char

Observação: Esse “problema” ocorre tanto com a função `scanf` como com a função `getchar`!



Caracteres - char

Existe alguma solução para esse problema?

Existe alguma solução para esse problema?

Resposta: **SIM.** A solução é bastante simples. Consiste em colocar um espaço em branco imediatamente antes do %c a partir do segundo scanf. O espaço deve estar dentro da *string* do formato.

Nota: O espaço em branco dentro de um scanf pede a essa função para **ler e ignorar** todo os Espaços em Branco, *New Lines* e *Tabs* que por ventura possa encontrar na memória *buffer* do teclado.

Caracteres - char

Exemplo: Consideremos o exemplo abaixo:

```
1: #include <stdio.h>
2:
3: void main()
4: {
5:     char ch1, ch2;
6:     printf("Introduza um Caractere: ");
7:     scanf("%c", &ch1);
8:     printf("Introduza um outro Caractere: ");
9:     scanf(" %c", &ch2);
10:    printf("Os caracteres introduzidos foram
11:          \'%c\' e \'%c\'\n", ch1, ch2);
}
```

Há uma outra solução

Uma outra maneira de resolver o problema é por meio do uso da função `fflush(stdin)`, acrescentando `fflush(stdin);` antes de cada um dos **scanf a partir do segundo**.

Exemplo: Crie outro código e ao invés de colocar `scanf(" %c", &ch2);` (espaço na *string* de formato), acrescente uma linha anterior e coloque o comando `fflush(stdin);`.

A íntima relação entre caracteres e inteiros

Ao contrário da maioria das linguagens de programação, os caracteres em C não são mais do que valores inteiros guardados em um único *byte*. Dessa forma, todas as operações sobre inteiros vistas anteriormente podem ser aplicadas ao tipo **char**.

Depois que declaramos uma variável do tipo **char** existem diversas formas de colocar um caractere nessa variável. Suponhamos que queremos criar uma variável **ch** do tipo **char** e queremos colocar no seu espaço de memória o caractere 'A'.

Exercício: Crie quatro programa que inicializa a variável caractere **ch** nas formas: **ch = 'A'** (primeiro), **ch = 65** (segundo), **ch = '\101'** (terceiro), **ch = '\x41'** (quarto). Imprima o resultado da seguinte forma:

Número Hexadecimal

```
printf("O caractere \'%c\' tem o ASCII Nu. %d\n",
ch, ch);
```

Números em hexadecimal estão representados na base 16. É muito comum em C escrever um inteiro nessa base. Os números 10, 11, 12, 13, 14 e 15 são representados na base hexadecimal pelas letras A, B, C, D, E e F.

Tabela: Correspondências entre base decimal para base hexadecimal.

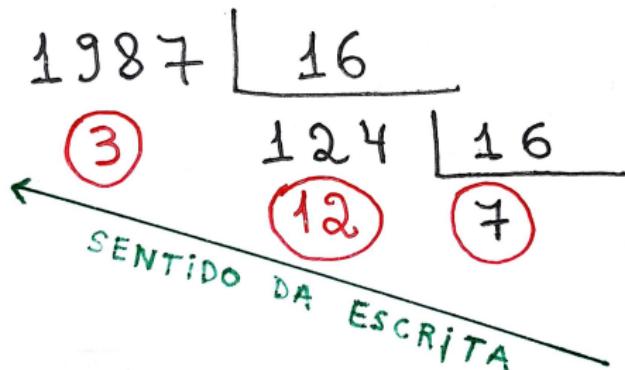
Decimal	0	1	2	...	8	9	10	11	12	13	14	15
Hexadecimal	0	1	2	...	8	9	A	B	C	D	E	F

Número Hexadecimal

Exemplo: Convertendo o número 1987 (número na base 10) para a base hexadecimal.

Número Hexadecimal

Exemplo: Convertendo o número 1987 (número na base 10) para a base hexadecimal.



Observação: Escrevendo os números da direita para esquerda temos os valores 7 12 3. Porém, note que 12 equivale em notação hexadecimal à letra C. Assim, o número **1987** representado em hexadecimal é **7C3**.

Número Hexadecimal

Exemplo: Convertendo número hexadecimal para a base decimal.

Número Hexadecimal

Exemplo: Convertendo número hexadecimal para a base decimal.

$$\begin{array}{r} 7 \ C \ 3 \\ \downarrow \quad \downarrow \quad \downarrow \\ 3 \cdot 16^0 = 3 \\ 12 \cdot 16^1 = 192 \\ 7 \cdot 16^2 = 1792 \\ \hline \Sigma = 1987 \end{array}$$

Exercício: Converta os números 1243, 1143, 20321, 65, 14 para a base hexadecimal.

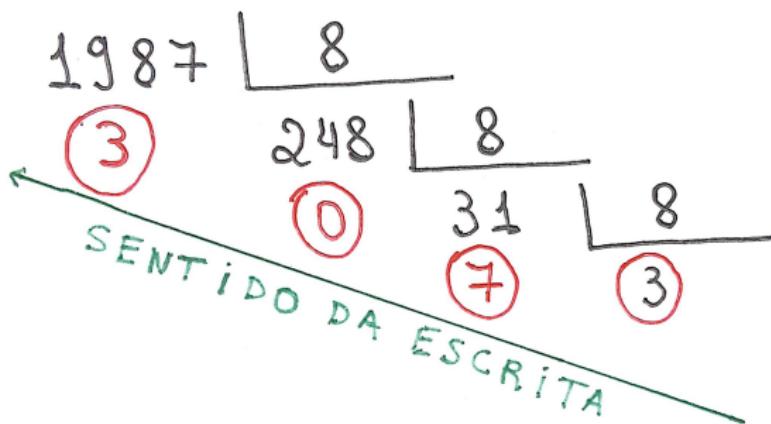
Exercício: Selecione 20 números com representação na base 10 da tabela <https://goo.gl/couXal> e converta-os para hexadecimal. Faça o processo inverso, converta-os de hexadecimal para decimal.

Número Octal

Chamamos de números octal aqueles que são escritos na base 8. Abaixo segue um exemplo de conversão de um número na base octal.

Número Octal

Chamamos de números octal aqueles que são escritos na base 8. Abaixo segue um exemplo de conversão de um número na base octal.



Assim, o valor **1987** para a base octal é **3703**. O número na base octal sempre será maior que na base decimal.

Número Octal

Exemplo: Convertendo o número **3073** (base octal) para a base decimal.

Número Octal

Exemplo: Convertendo o número **3073** (base octal) para a base decimal.

$$\sum = 1987$$
$$3 \cdot 8^0 = 3$$
$$0 \cdot 8^1 = 0$$
$$7 \cdot 8^2 = 448$$
$$3 \cdot 8^3 = 1536$$

Exercício: Selecione 20 números com representação na base 10 da tabela <https://goo.gl/couXal> e converta-os para octal. Faça o processo inverso, converta-os de octal para decimal.

A íntima relação entre caracteres e inteiros

Voltando ao último exercício: Crie quatro programa que inicializa a variável caractere ch nas formas: ch = 'A' (primeiro), ch = 65 (segundo), ch = '\101' (terceiro), ch = '\x41' (quarto).

```
1: #include <stdio.h>
2:
3: void main()
4: {    char ch;
5:     printf("Introduza um Caractere: ");
6:     scanf("%c", &ch);
7:     printf("O caractere \'%c\' tem o ASCII Nu. %d\n",
8:            ch, ch);
9: }
```

Casting

Muito Importante

Sempre que em uma variável ou expressão temos um valor de um determinado tipo e queremos modificar o tipo desse valor, alterando-o para um tipo maior ou para um tipo mais baixo, podemos indicar o tipo ao qual queremos “promover” esse valor colocando o tipo pretendido entre parênteses antes do valor.

No exemplo anterior o código `printf("O caractere \'%c\' tem o ASCII Nu. %d\n", ch, ch);` passamos `ch` (terceiro argumento passado à função `printf`) que é uma variável do tipo **char** e foi impresso um inteiro na tela, isto é, o *casting* foi automático.

Observação: Evite *casting* automático. **Na maioria dos casos**, a conversão entre tipos não é automática.

Casting

Nota: Os formatos de leitura e escrita de números hexadecimal é %x ou %X. No caso de números octal utilizamos o formato %o.

Observação: O motivo que faz com que o *casting* seja automático entre caracteres e inteiros é o fato de que existe uma correspondência entre inteiros e caracteres estabelecidos nos diversos padrões C. Novamente, essa relação pode ser observada na tabela ASCII <https://goo.gl/couXal>. Sendo assim, por via de regra, sempre utilize *casting* conscientes.

Casting

Exercício: Escreva um programa que solicite um inteiro (entre 0 e 255) ao usuário e mostre o inteiro seguinte e o caractere correspondente.

Casting

```
1: #include <stdio.h>
2:
3: void main()
4: {    int num;
5:     printf("Introduza um Inteiro: ");
6:     scanf("%d", &num);
7:     printf("Foi introduzido %d cujo caractere =
8:             \'%c\'\n", num, (char) num);
9:     printf("O caractere seguinte = \'%c\' tem o
10:           ASCII numero %d\n", (char) (num+1), num+1);
11:}
```

Outros Detalhes (Inteiro e Caractere)

Muita Atenção

Apesar que é possível fazer conversões de inteiros para caracteres e vice-versa, a leitura de inteiro com formato %c leva a resultados completamente inesperados e bastantes perigosos. Assim, **nunca leia inteiro com o formato %c.**

A razão de não fazermos isso se deve ao fato de que quando se pede para ler um caractere (%c) a função scanf irá reservar apenas um *byte* na memória. Ao colocar o caractere lido em um inteiro, apenas um dos *bytes* do inteiro será alterado. Os demais **bytes** ficam exatamente como estavam antes da chamada à função scanf.

Outros Detalhes (Inteiro e Caractere)

```
1: #include <stdio.h>
2:
3: void main()
4: {
5:     int num = 1000; /*Colocando lixo em num */
6:     printf("Introduza um Caractere: ");
7:     scanf("%c", &num);
8:     printf("O valor de num = %d cujo caractere
9:             = \'%c\'\n", num, (char) num);
10:}
```

Introduza o caractere A quando for solicitado um caractere.

Outros Detalhes (Inteiro e Caractere)

```
1: #include <stdio.h>
2:
3: void main()
4: {
5:     int num = 1000; /*Colocando lixo em num */
6:     printf("Introduza um Caractere: ");
7:     scanf("%c", &num);
8:     printf("O valor de num = %d cujo caractere
9:             = \'%c\'\n", num, (char) num);
10:}
```

Introduza o caractere A quando for solicitado um caractere. **Por que o num = 833?**

Outros Detalhes (Inteiro e Caractere)

Resposta: O número 1000 em binário é 11111 01000. Como um inteiro é armazenado em 2 *bytes* temos que ele ficará na memória na forma que segue:

```
00000011 11101000
```

Ao ler o caractere ‘A’ (com %c) cujo valor inteiro na tabela ASCII é 65 e jogarmos a informação dentro de num que tinha como lixo o valor 1000 é substituído no byte mais a direita da variável num a representação binária de 65 que é 01000001.

Assim, temos o número 00000011 01000001. Convertendo para decimal temos o valor **833**.

Outros Detalhes (Inteiro e Caractere)

Vamos ver algo mais estranho com o código que segue:

```
1: #include <stdio.h>
2: void main()
3: {
4:     char ch1 = 'X', ch2 = 'Y';
5:     printf("Introduza um Inteiro: ");
6:     scanf("%d", &ch2);
7:     printf("O valor de ch1 = \'%c\' e ch2 = \'%c\'\n",
8:            ch1, ch2);
9: }
```

Nesse exemplo acima, ao ser pedido a entrada de um número inteiro, introduza o valor **16706**.

Outros Detalhes (Inteiro e Caractere)

Vamos ver algo mais estranho com o código que segue:

```
1: #include <stdio.h>
2: void main()
3: {
4:     char ch1 = 'X', ch2 = 'Y';
5:     printf("Introduza um Inteiro: ");
6:     scanf("%d", &ch2);
7:     printf("O valor de ch1 = \'%c\' e ch2 = \'%c\'\n",
8:            ch1, ch2);
9: }
```

Nesse exemplo acima, ao ser pedido a entrada de um número inteiro, introduza o valor **16706**. Por que os caracteres armazenados em ch1 e ch2 foram A e B?

Outros Detalhes (Inteiro e Caractere)

Inicialmente as variáveis do tipo **char** (`ch1` e `ch2`) correspondem segundo a tabela ASCII aos números binários 01011000 e 01011001, respectivamente.

Suponha por azar que `ch1` e `ch2` sejam armazenadas em posições consecutivas na memória. **Normalmente isso acontece!**

Daí, no código acima lemos um número inteiro (**16706**) que ocupa 2 *bytes* na memória mas a variável `ch2` armazena apenas 1 *byte*. Dessa forma, o restante das informações irão extrapolar para a variável `ch1`.

Outros Detalhes (Inteiro e Caractere)

Inicialmente com ch1 e ch2 ocupando posições consecutivas na memória, temos a seguinte sequência de *bits*:

01011000 01011001

Ao colocar o número 19706 que em representação binária é 10000001 01000010, geramos uma nova sequência:

01000001 01000010

O primeiro *byte* (01000001) corresponde ao inteiro 65 na tabela ASCII e refere-se ao caractere 'A'. De forma análoga, o segundo *byte* (01000010) corresponde ao inteiro 66 na mesma tabela refere-se ao caractere 'B'.

Tipos de Dados em C

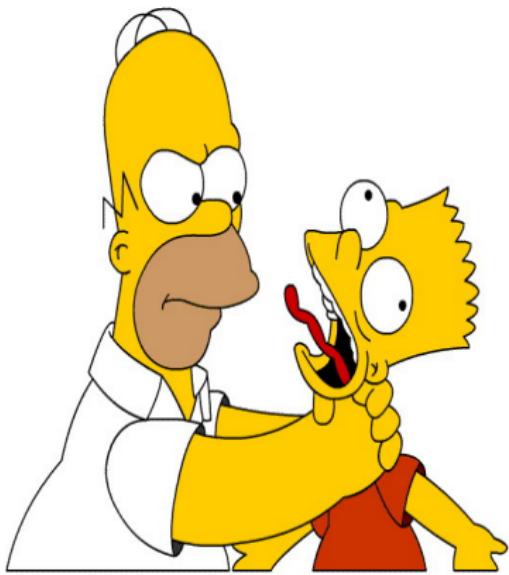
IMPORTANTE: Para não ter surpresas desagradáveis, jamais se deve realizar leitura/escrita de variáveis de um determinado tipo usando um formato de leitura/escrita que não corresponda a esse tipo!

Alguns casos poderá dar certo devido haver, em algumas situações, *casting* automático. Porém, a boa prática de programação exige os formatos de tipos corretos.

Tipos de Dados em C

IMPORTANTE: Para não ter surpresas desagradáveis, jamais se deve realizar leitura/escrita de variáveis de um determinado tipo usando um formato de leitura/escrita que não corresponda a esse tipo!

Alguns casos poderá dar certo devido haver, em algumas situações, *casting* automático. Porém, a boa prática de programação exige os formatos de tipos corretos.



Tipos de Dados em C

Exercício: Escreva um programa em C que peça ao usuário dois números inteiros e apresente o resultado da realização das operações aritméticas tradicionais.

Exercício: Escreva um programa em C que solicite um determinado número de segundos e, em seguida, indique quantas hora, minutos e segundos esse valor representa.

Exercício: Escreva um programa que solicite ao usuário uma determinada data e mostre em tela no formato *dd/mm/aaaa*.

Exercício: Escreva um programa que solicite ao usuário uma determinada data no formato *aaa-mm-dd* e mostre em tela e em seguida apresente a data no formato *dd/mm/aaaa*.

Programação Aplicada à Estatística

Pedro Rafael Diniz Marinho

Universidade Federal da Paraíba

2021.2

Valores Lógicos - Verdadeiro e Falso

Até o presente momento, todos os programas que construímos estão adaptados a um mundo perfeito, isto é, ao mundo sem existência de erros, condições ou qualquer tipo de variação.

Assim sendo, o que víhamos fazendo era construir programas em que as instruções seguem-se umas às outras, seguindo sempre a mesma ordem de execução, independentemente do valor de entrada.

Por exemplo, ao se vestir e vir para Universidade, muitos consideram as condições do tempo e daí decidem que tipo de roupa irá vestir.

Valores Lógicos - Verdadeiro e Falso

Fazendo uma analogia ao problema de se vestir e vir à Universidade, os programas que fazíamos eram bastante diretos como o algoritmo à esquerda. Já o algoritmo da direita é um pouco mais interessante mesmo que não necessariamente venha a ser perfeito.

Algoritmo não muito legal:

- Vestir Camiseta
- Vestir Casaco
- Vestir Calça
- Calçar Sapatos
- Pentear

Algoritmo mais coerente:

- Vestir Camisa
- Se frio, então vestir Casaco**
- Vestir Calças
- Calçar Sapatos
- Pentear

Valores Lógicos - Verdadeiro e Falso

No mundo que vivemos sempre temos que tomar decisões e a depender das informações que temos, determinadas ações específicas serão realizadas. C é capaz de checar condições e tomar decisões em meio dessa checagem.

Valores Lógicos - Verdadeiro e Falso

No mundo que vivemos sempre temos que tomar decisões e a depender das informações que temos, determinadas ações específicas serão realizadas. C é capaz de checar condições e tomar decisões em meio dessa checagem.



Valores Lógicos - Verdadeiro e Falso

Como vimos anteriormente C possui quatro tipos básicos de dados (**int**, **float**, **double** e **char**) e o tipo ponteiro. Porém, não existe um tipo específico para representar um valor lógico (**Verdadeiro** e **Falso**).

Em outras linguagem de programação esses tipos podem existir

Por exemplo, na linguagem R muito embora trate-se de uma linguagem dinâmica, é possível passar para uma variável os valores booleanos **TRUE** ou **FALSE**. Um outro exemplo é a linguagem Pascal, em que existe o tipo **BOOLEAN** o qual uma variável com esse tipo também poderá receber **TRUE** ou **FALSE**.

Valores Lógicos - Verdadeiro e Falso

Muito Importante

Pelo fato de em C não existir nenhum tipo de dados para armazenar valores lógicos, o valor lógico **FALSO** é representado por 0 (**ZERO**). Já, tudo aquilo que for diferente de 0 (**ZERO**) representará o valor lógico **VERDADEIRO**.

Exemplo:

Falso : 0

Verdade : 2, -7, 123.447, 0.000001

Valores Lógicos - Verdadeiro e Falso

Lembre-se:

O valor lógico **VERDADE** em C não é o valor 1, mas tudo que for diferente de **ZERO**. Alguns materiais na internet costumam explicar de forma que para quem está lendo é natural pensar em 0 como sendo **FALSO** e 1 como sendo **VERDADE**.

Observação: Os operadores lógicos são resultados de avaliações realizadas pelos operadores relacionais. Tais operadores são: ==, >, <, >=, <= e !=. Maiores detalhes encontram-se na tabela que segue.

Operadores Relacionais

Em C existem um conjunto de seis operadores relacionais, os quais podem ser usados na avaliação de expressões. O seu objetivo consiste no estabelecimento de relações entre operandos.

Tabela: Operadores relacionais utilizados pela linguagem C.

Operador	Nome	Exemplo	Significado do Exemplo
<code>==</code>	Igualdade	<code>a == b</code>	<code>a</code> é igual a <code>b</code> ?
<code>></code>	Maior que	<code>a > b</code>	<code>a</code> é maior que <code>b</code> ?
<code>>=</code>	Maior ou Igual que	<code>a >= b</code>	<code>a</code> é maior ou igual a <code>b</code> ?
<code><</code>	Menor que	<code>a < b</code>	<code>a</code> é menor que <code>b</code> ?
<code><=</code>	Menor ou Igual que	<code>a <= b</code>	<code>a</code> é menor ou igual a <code>b</code> ?
<code>!=</code>	Diferente de	<code>a != b</code>	<code>a</code> é diferente de <code>b</code> ?

Operadores Relacionais

Importante

Apesar que qualquer valor diferente de **ZERO** tem o valor lógico **VERDADE**, uma expressão que contenha um operador relacional devolve sempre como resultado o valor lógico **VERDADE** (1) ou **FALSO** (0).

Porém, pela última vez, qualquer valor diferente de **ZERO** representa o valor lógico **VERDADE**.

Exercício: Implemente um programa que solicite ao utilizador dois inteiros e, em seguida, imprima na tela o resultado de todas as operações relacionais de C aos inteiros lidos.

Operadores Relacionais

```
1: #include <stdio.h>
2:
3: void main()
4: {
5:     int x,y;
6:     printf("Introduza 2 inteiros: ");
7:     scanf("%d%d", &x, &y);
8:     printf("O resultado de %d == %d : %d\n",x,y,x==y);
9:     printf("O resultado de %d > %d : %d\n",x,y,x>y);
10:    printf("O resultado de %d < %d : %d\n",x,y,x<y);
11:    printf("O resultado de %d >= %d : %d\n",x,y,x>=y);
12:    printf("O resultado de %d <= %d : %d\n",x,y,x<=y);
13:    printf("O resultado de %d != %d : %d\n",x,y,x!=y);
14: }
```

Operadores Relacionais

Os operadores relacionais podem ser aplicados a expressões simples ou complexas com no exemplo que segue:

Exemplo:

$4 \geq 2$

$x \leq y$

$(x+y) < (x*2+y*(-4))$

Atenção

Um erro bastante frequente em programação é a troca do operador relacional $==$ pelo operador de atribuição $=$. O operador relacional $==$ verifica se duas expressões são iguais, enquanto o operador de atribuição $=$ é utilizado para atribuir valores a uma variável em C.

if-else

Muitas das operações relacionais que utilizamos é em funções de controle de fluxo. Em C, a instrução if-else é uma das instruções para controlar o fluxo do programa.

Instruções de Controle de Fluxo

Tais instruções permitem indicar sob quais circunstâncias o programa irá executar determinada instrução ou conjunto de instruções. A sintaxe geral é:

```
if (condição)
    instrução1;
[ else instrução2; ]
```

Tudo que está entre colchetes poderá ser utilizado ou não, isto é, o código dentro dos colchetes é opcional a depender do que se pretende programar.

if-else

Com as instruções de controle de fluxo if-else permitimos que parte do nosso programa possa a vir a nunca ser executada. Para tanto, basta que a condição necessária para a sua execução nunca se verifique.

A instrução if-else funciona da seguinte maneira:

- A condição é avaliada;
- Se o resultado da condição for verdadeiro, executa a instrução1;
- Se o resultado da condição for falso, executa a instrução2 (**caso exista o else**).

Observação: Jamais coloque ponto-e-vírgula (;) depois do if ou depois do else. Apenas as instruções em seu interior são terminadas por ponto-e-vírgula.

if-else

Exercício: Implemente um programa que indique se um número é não-negativo (≥ 0) ou negativo.

if-else

Exercício: Implemente um programa que indique se um número é não-negativo (≥ 0) ou negativo.

```
1: #include <stdio.h>
2:
3: void main()
4: {
5:     int x;
6:     printf("Introduza um N%c: ", 167);
7:     scanf("%d", &x);
8:     if (x>=0)
9:         printf("Número Positivo\n", 163);
10:    else
11:        printf("Número Negativo\n", 163);
12: }
```

if-else

Exercício: Implemente um programa que indique se o inteiro lido é zero ou não.

```
1: #include <stdio.h>
2:
3: void main()
4: {
5:     int x;
6:     printf("Introduza um N%c: ", 167);
7:     scanf("%d", &x);
8:     if (x != 0)
9:         printf("%d n%co %c zero!!!\n", x, 198, 130);
10:    else
11:        printf("%d %c igual a zero!!!\n", x, 130);
12:}
```

if-else

Exemplo: Uma outra forma de implementar o código no exemplo logo acima é:

if-else

Exemplo: Uma outra forma de implementar o código no exemplo logo acima é:

```
1: #include <stdio.h>
3: void main()
4: {
5:     int x;
6:     printf("Introduza um N%c: ", 167);
7:     scanf("%d", &x);
8:     if (x)
9:         printf("%d n%co %c zero!!!\n", x, 198, 130);
10:    else
11:        printf("%d %c igual a zero!!!\n", x, 130);
12:}
```

Exercício: Indique a diferença e justifique o motivo de dar certo.

if-else

Dica:

Em C, o valor de uma variável ou de uma constante pode ser aproveitado pelo programador como valor lógico, utilizando-o como **FALSO** (caso o valor seja 0) ou **VERDADE** (caso o seu valor seja diferente de 0).

if-else

Dica:

Em C, o valor de uma variável ou de uma constante pode ser aproveitado pelo programador como valor lógico, utilizando-o como **FALSO** (caso o valor seja 0) ou **VERDADE** (caso o seu valor seja diferente de 0).

Observação: Em geral, muitos programadores mais experientes da linguagem C gostam de escrever um código que mais ninguém comprehende, recorrendo a algumas características e peculiaridades da própria linguagem. No entanto, é comum que o código fique muito compacto e confuso de modo que os próprios programadores têm dificuldades de comprehendê-lo.

if-else

Dica:

Um programa deve estar escrito de tal maneira que um leve passar de olhos pelo código fonte indique ao leitor, sem sombra de dúvidas, aquilo que a aplicação deve fazer. Comentário também poderão ajudar bastante na compreensão.

Exercício: Implemente um programa que adicione R\$ 1000 reais ao salário de um indivíduo, caso este seja inferior a R\$ 100.000,00 reais.

if-else

```
1: #include <stdio.h>
2:
3: void main()
4: {
5:     float salario;
6:     printf("Digueite o Sal%crio: ", 160);
7:     scanf("%f", &salario);
8:     if(salario < 100000)
9:         salario = salario + 1000;
10:    printf("Sal%crio Final: %.2f\n", 160, salario);
11:}
```

if-else

```
1: #include <stdio.h>
2:
3: void main()
4: {
5:     float salario;
6:     printf("Digueite o Sal%crio: ", 160);
7:     scanf("%f", &salario);
8:     if(salario < 100000)
9:         salario = salario + 1000;
10:    printf("Sal%crio Final: %.2f\n", 160, salario);
11:}
```

Nota: Para que a saída do Salário Final fique formatada e não seja imprimido um número com muitas casas decimais, a parte fracionaria da variável salario foi limitada por dois dígitos significativos utilizando %.2f.

if-else

E se queremos colocar mais instruções no if-else?

Resposta: Nessas situações, deveremos utilizar **Bloco de Instruções** que podem seguir o if e/ou o else. Sembre-se que um bloco de instruções é tudo aquilo compreendido entre { }.

O compilador C entenderá que todas as instruções contidas no **Bloco de Instruções** como uma única instrução. Sendo assim, o **Bloco** é um conjunto de duas ou mais instruções delimitadas por chaves.

Observação: Todo o lugar em que se poderá colocar uma instrução simples poderá ser colocado um bloco de instruções.

Nota: Depois de um **Bloco** não é necessário colocar ponto-e-vírgula (;).

if-else

Exercício: Escreva um programa que leia dois números e os apresente por ordem crescente.

if-else

Exercício: Escreva um programa que leia dois números e os apresente por ordem crescente.

```
#include <stdio.h>
void main()
{
    int x, y, tmp;
    printf("Introduza dois N\%cs: ", 167);
    scanf("%d %d", &x, &y);
    if (x > y)
    {
        tmp = x;
        x = y;
        y = tmp;
    }; /* Colocar esse ponto-e-virgula não é obri. */
    printf("%d %d\n", x, y);
}
```

Instruções if-else Encadeadas

Em diversas situações não é suficiente um único teste de condições para se tomar uma decisão. Sendo assim, pode ser necessário se testar mais de uma condição.

Exercício: Escreva um programa em C que solicite um salário ao utilizador e mostre o importo a pagar.

- Se o salário for negativo ou zero mostre o erro respectivo (coloque uma mensagem de erro);
- Se o salário for maior que 1000, paga 10% de imposto, se não apenas 5%.

Instruções if-else Encadeadas

```
1: #include <stdio.h>
2:
3: void main()
4: {
5:     float salario;
6:     printf("Digite o sal%crio: ", 160);
7:     scanf("%f", &salario);
8:     if (salario <=0)
9:         printf("Sal%crio: Valor Inv%clido\n",
10:                160, 160);
11:     if (salario > 1000)
12:         printf("Imposto = %.2f\n", salario*0.10);
13:     else
14:         printf("Imposto = %.2f\n", salario*0.05);
```

Instruções if-else Encadeadas

Exercício: Digite um valor negativo. Observe que algo estranho ocorrerá. Por que isso está ocorrendo? Explique o funcionamento desse programa de modo a entendermos o erro. Modifique o programa de modo a solucionar esse problema.

Instruções if-else Encadeadas

```
1: #include <stdio.h>
2:
3: void main()
4: {
5:     float salario;
6:     printf("Digite o sal%crio: ", 160);
7:     scanf("%f", &salario);
8:     if (salario <= 0)
9:         printf("Sal%crio: Valor Inv%clido\n",
10:                160, 160);
11:    else
12:        if (salario > 1000)
13:            printf("Imposto = %.2f\n", salario*0.10);
14:        else
15:            printf("Imposto = %.2f\n", salario*0.05);
```

Instruções if-else Encadeadas

Muito Importante

Sempre que existam instruções if-else encadeadas, cada componente else pertence sempre ao último if (que ainda não tenha um else associado).

Observação: Quando temos if-else encadeados, nos ajudará bastante manter uma boa indentação do código fonte. Caso contrário, em um código relativamente grande que apresente muitas instruções if-else encadeadas, será fácil não entender quais instruções if e else estão relacionadas.

Nota: No código abaixo veja que foi colocado void como argumento da função main. Em todos os códigos trabalhados anteriormente, isso poderia ser feito uma vez que, em geral, para qualquer função que não tem argumentos para serem passados, esta poderá receber argumento do tipo void.

Instruções if-else Encadeadas

Exemplo:

```
1: #include <stdio.h>
2: void main(void)
3: { int a,b;
4:   printf("Introduza dois números: ");
5:   scanf("%d%d",&a,&b);
6:   if (a >= 0)
7:     if (b > 10)
8:       printf("B é muito grande\n");
9:     else
10:    printf("A tem um valor negativo\n");
}
```

Tente digitar quando for solicitado ao executar o programa os valores 1 e 9. É fácil perceber que o programa não está informando corretamente o sinal de a.

Instruções if-else Encadeadas

Exemplo: Para que o código funcione corretamente, isto é, para que o else esteja associado ao primeiro if deixe explícito isto com o uso de chaves {}.

```
1: #include <stdio.h>
2: void main(void)
3: { int a,b;
4:   printf("Introduza dois números: ");
5:   scanf("%d%d",&a,&b);
6:   if (a >= 0)
7:     { /* Repare nas chaves */
8:       if (b > 10)
9:         printf("B é muito grande\n");
10:      } /* Repare nas chaves */
11:    else
12:      printf("A tem um valor negativo\n");}
```

Instruções if-else Encadeadas

Regra de Bolso

Se um conjunto de instruções if encadeadas alguma delas não necessitar de else, deve-se colocar essa instrução entre chaves para que o else de algum outro if mais exterior não seja associado erradamente a este if.

Exercício: Crie um programa que solicite 3 notas separadas por vírgula ao aluno e informe se o aluno foi reprovado ou foi aprovado. O programa deverá imprimir ao usuário do programa as notas digitadas, a média final e a situação do aluno (aprovado ou reprovado). Não é necessário tratar a situação do aluno que tem direito à ir para prova final. Veremos isso adiante quando for apresentado os operadores lógicos.

Operadores Lógicos

Existem diversas situações em que uma simples condição não é suficiente para tomar uma decisão, sendo por isso necessária a interligação de duas ou mais condições. Para tal, a maioria das linguagens de programação coloca à disposição um conjunto de operadores lógicos, os quais funcionam da mesma forma que os operadores aritméticos, mas aplicados a valores lógicos.

Tabela: Operadores lógicos em C.

<code>&&</code>	AND (E lógico)	$x \geq 1 \text{ } \&\& \text{ } x \leq 19$
<code> </code>	OR (OU lógico)	$x == 1 \text{ } \text{ } x == 2$
<code>!</code>	NOT (Negação lógica)	<code>!x</code>

Operadores Lógicos

Importante

Os operadores lógicos são operadores binários, com exceção do operador ! que é unário.

Tabela: Operadores lógicos em C.

<code>&&</code>	<code>(cond1 && cond2)</code>	
<code> </code>	<code>(cond1 cond2)</code>	
<code>!</code>	<code>(!cond) (Negação lógica)</code>	

Operadores Lógicos

Importante

Os operadores lógicos são operadores binários, com exceção do operador ! que é unário.

Tabela: Operadores lógicos em C.

&&	(cond1 && cond2)	♠
	(cond1 cond2)	♣
!	(!cond) (Negação lógica)	♦

- ♠ Verdade se ambas as condições forem verdadeiras, Falso no caso contrário.

Operadores Lógicos

Importante

Os operadores lógicos são operadores binários, com exceção do operador ! que é unário.

Tabela: Operadores lógicos em C.

&&	(cond1 && cond2)	♠
	(cond1 cond2)	♣
!	(!cond) (Negação lógica)	♦

- ♠ Verdade se ambas as condições forem verdadeiras, Falso no caso contrário.
- ♣ Verdade se alguma das condições for verdadeira e Falso se ambas forem falsas.

Operadores Lógicos

Importante

Os operadores lógicos são operadores binários, com exceção do operador ! que é unário.

Tabela: Operadores lógicos em C.

&&	(cond1 && cond2)	♠
	(cond1 cond2)	♣
!	(!cond) (Negação lógica)	♦

- ♠ Verdade se ambas as condições forem verdadeiras, Falso no caso contrário.
- ♣ Verdade se alguma das condições for verdadeira e Falso se ambas forem falsas.
- ♦ Verdade se cond for falsa. Falso se cond for Verdadeira.

Operadores Lógicos

Exercício: Crie um programa que solicite 3 notas separadas por vírgula ao aluno e informe se o aluno foi aprovado, reprovado ou está na final. O programa deverá imprimir em tela as três notas digitadas pelo aluno, sua média, a sua situação (aprovado, reprovado ou na final) e caso o aluno esteja na final, informe também quantos pontos ele precisa para passar na disciplina. Caso o aluno seja reprovado, não esqueça de passar a informação educadamente dando-o uma palavra de apoio 😊 .

Operadores Lógicos

Exercício: Escreva um programa que calcule o Salário Bruto, o Salário Líquido e o Imposto a pagar seguindo a seguinte regra de tributação:

Tabela: Regra de tributação de salário.

Salário	Taxa
<1000	5%
≥ 1000 e <5000	11%
≥ 5000	35%

Precedência dos Operadores

Ao utilizarmos diversos operadores em uma mesma expressão, o compilador terá que decidir qual a ordem da realização das operações. O compilador decide tais ordens por meio da Tabela de precedência de operadores. A Tabela abaixo apresenta a precedência dos operadores em C. A seta indica que quanto mais elevado estiver o operador, maior será sua precedência.



Operador			
< <= > >=			
== !=			
&&			
?:			

Precedência dos Operadores

Exercício: Suponha que $x=7$ e $y=0$. Além disso seja

```
if (x != 10 || y > 1 && y < 10)
```

O compilador de C decidirá em executar o if? Qual o resultado do teste de condição feito pelo compilador?

Importante

Em C, o operador `&&` tem maior precedência que o operador `||`.

Exercícios

Exercício: Crie um programa em C que solicite três números ao usuário e apresente em tela o número maior e o número menor.

Exercício: Escreva um programa em C que recebe um inteiro e diga se é par ou ímpar. **Dica:** Utilize o operador matemático %.

Exercício: Para doar sangue é necessário ter entre 18 e 67 anos. Além disso, a pessoa só poderá doar sangue se não tiver nenhuma doença que comprometa a qualidade do sangue doado. Construa um programa em C que pergunte a idade de uma pessoa e se ela tem alguma doença que possa comprometer a doação. O programa deverá informar se a pessoa é uma possível doadora ou não, a depender das respostas passadas ao programa.

Exercícios

Exercício: Escreva um programa que leia as medidas dos lados de um triângulo e escreva em tela se ele é equilátero, isósceles ou escaleno.

Exercícios

Exercício: Escreva um programa que leia as medidas dos lados de um triângulo e escreva em tela se ele é equilátero, isósceles ou escaleno. Lembrando que:

- ① Triângulo Equilátero: aquele que possui os três lados iguais.

Exercícios

Exercício: Escreva um programa que leia as medidas dos lados de um triângulo e escreva em tela se ele é equilátero, isósceles ou escaleno. Lembrando que:

- ① Triângulo Equilátero: aquele que possui os três lados iguais.
- ② Triângulo Isósceles: aquele que possui dois lados iguais.

Exercícios

Exercício: Escreva um programa que leia as medidas dos lados de um triângulo e escreva em tela se ele é equilátero, isósceles ou escaleno. Lembrando que:

- ① Triângulo Equilátero: aquele que possui os três lados iguais.
- ② Triângulo Isósceles: aquele que possui dois lados iguais.
- ③ Triângulo Escaleno: aquele que possui três lados diferentes.

Exercícios

Exercício: Escreva um programa em C que solicite três notas de um aluno e informe a média das três avaliações. Além disso, o programa deverá informar se o aluno foi aprovado, reprovado ou terá direito a fazer uma prova final. Em caso do aluno ir para final, informe quanto ele deverá obter na final para se considerar aprovado na disciplina.

Exercícios

Exercício: Escreva um programa em C que solicite três notas de um aluno e informe a média das três avaliações. Além disso, o programa deverá informar se o aluno foi aprovado, reprovado ou terá direito a fazer uma prova final. Em caso do aluno ir para final, informe quanto ele deverá obter na final para se considerar aprovado na disciplina.

Regra: O aluno será considerado aprovado se tiver a média aritmética das três avaliações maior ou igual à sete. Se a média for menor que 4, o aluno será reprovado sem o direito de fazer o exame final. Já no caso da média aritmética esteja no intervalo [4, 7) o aluno poderá fazer um exame final. Nessa última situação, a nota final assumirá peso 4 e a média aritmética das três primeiras avaliações assumirá o peso 6. Se o aluno obter uma média ponderada maior ou igual à 5.0 ele estará aprovado.

Exercícios

Exercício: Uma empresa abriu uma linha de crédito para os funcionários. O valor da prestação não pode ultrapassar 30% do salário. Faça um programa que receba o salário, o valor do empréstimo e o número de prestações e informe se o empréstimo pode ser concedido. Nenhum dos valores informados pode ser zero ou negativo. Sendo assim, o programa deverá alertar o usuário que cometer esse equívoco.

Operador Condicional ?

O operador condicional é o único operador ternário em C, o que indica que este espera três argumentos.

```
condição ? expressão1 : expressão2
```

Funcionamento:

Operador Condicional ?

O operador condicional é o único operador ternário em C, o que indica que este espera três argumentos.

```
condição ? expressão1 : expressão2
```

Funcionamento:

- A condição é avaliada;

Operador Condicional ?

O operador condicional é o único operador ternário em C, o que indica que este espera três argumentos.

```
condição ? expressão1 : expressão2
```

Funcionamento:

- A condição é avaliada;
- Se o resultado for Verdade, o resultado de toda a expressão é o valor devolvido por expressão1.

Operador Condisional ?

O operador condicional é o único operador ternário em C, o que indica que este espera três argumentos.

```
condição ? expressão1 : expressão2
```

Funcionamento:

- A condição é avaliada;
- Se o resultado for Verdade, o resultado de toda a expressão é o valor devolvido por expressão1.
- Se o resultado for Falso, o resultado de toda a expressão é o valor devolvido por expressão2.

Operador Condisional ?

Exercício: Implemente um programa que calcule os aumentos de ordenados para o corrente ano. Se o ordenado for maior que 1000 deve ser aumentado em 5%, se não, deve ser aumentado em 7%. Crie dois códigos, em que o primeiro o programa é escrito com o uso de `if` e `else`. Já o segundo, deverá usar o Operador Condisional `?`.

Operador Condicional ?

Programa 1:

```
1: #include <stdio.h>
2:
3: void main(void)
4: {
5:     float salario;
6:
7:     printf("Qual o Salário: ");
8:     scanf("%f",&salario);
9:     if (salario > 1000)
10:         salario = salario * 1.05;
11:     else
12:         salario = salario * 1.07;
13:     printf("Novo Salário: %.2f\n", salario);
14: }
```

Operador Condicional ?

Programa 2:

```
1: #include <stdio.h>
2:
3: void main(void)
4: {
5:     float salario;
6:
7:     printf("Qual o Salário: ");
8:     scanf("%f",&salario);
9:     salario = salario > 1000 ? salario*1.05 :
10:                     salario*1.07;
11: }
```

Switch

Quando existem alguns conjuntos de condições possíveis para o mesmo valor, torna-se impraticável a utilização da instrução `if-else` pois o código torna-se confuso e extenso. Para resolver esses tipos de problemas, C fornece uma outra instrução que permite a seleção do código a executar, a partir de um conjunto de valores possíveis para uma determinada expressão.

Regra de Bolso

A instrução `switch` adapta-se particularmente à tomada de decisões em que o número de possibilidades é elevado (em geral, maior que 2, se não usa-se `if-else`).

Switch

Sintaxe:

```
switch (expressão)
{
    case constante_1 : instruções_1;
    case constante_2 : instruções_2;
    ...
    case constante_n : instruções_n;
    [ default: instruções; ]
}
```

Observação: O uso de **default: instruções;** como está dentro de colchetes, é opcional. Lembre-se que quando estamos escrevendo a sintaxe do uso de uma função ou código, convencionamos em aulas anteriores que tudo que está dentro de [] é opcional.

Switch

Sintaxe:

Importante

Na sintaxe do uso da função switch, expressão significa qualquer valor numérico dos tipos **char**, **int** ou **long**. O resultado da expressão é avaliada e comparada com as constantes de cada um case.

Switch

Funcionamento do switch:

Funcionamento do switch:

- Se o valor da expressão for igual a alguma das constantes que seguem os vários **case**, então são executadas as instruções que seguem o **case** correspondente.

Switch

Funcionamento do switch:

- Se o valor da expressão for igual a alguma das constantes que seguem os vários **case**, então são executadas as instruções que seguem o **case** correspondente.
- Se o valor da expressão não for igual a nenhuma das constantes apresentadas pelos **case**, então são executadas as instruções que seguem o **default**.

Switch

Funcionamento do switch:

- Se o valor da expressão for igual a alguma das constantes que seguem os vários **case**, então são executadas as instruções que seguem o **case** correspondente.
- Se o valor da expressão não for igual a nenhuma das constantes apresentadas pelos **case**, então são executadas as instruções que seguem o **default**.
- Como se pode observar e já foi mencionado, o **default** é opcional. No caso de não haver o **default** e o valor de **expressão** não ser igual a nenhum **case**, nada é executado. Sendo assim, o programa irá continuar na próxima instrução que segue o **switch**.

Switch

Importante

Em cada **case** do **switch** só há uma única constante (**char**, **int**, **long**).

Exercício: Implemente um programa em C que dada uma letra, o programa imprima na tela o estado civil do usuário fazendo uso das funções **if-else**. Depois implemente o mesmo programa em que faz uso **apenas** da função **switch**. Qual programa ficou mais confuso? Explique.

Switch

Primeiro Caso:

```
1: #include<stdio.h>
2:
3: void main()
4: {   char Est_Civil;
5:     puts("Informe o Estado Civil: ");
6:     Est_Civil = getchar();
7:     if (Est_Civil == 'S' || Est_Civil == 's')
8:         printf("Solteiro");
9:     else
10:        if (Est_Civil == 'C' || Est_Civil == 'c')
11:            printf("Casado");
12:        else
13:            if (Est_Civil == 'D' || Est_Civil == 'd')
14:                printf("Divorciado");
```

Switch

```
15:     else
16:         if (Est_Civil == 'V' || Est_Civil == 'v')
17:             printf("Viúvo");
18:         else
19:             printf("Estado Civil Inválido");
20:     printf("\n");
21: }
```

Switch

Segundo Caso:

```
1: #include <stdio.h>
2:
3: void main()
4: { char Est_Civil;
6:   printf("Qual o estado Civil: ");
7:   scanf(" %c", &Est_Civil); /* Ou usar getchar(); */
8:   switch(Est_Civil)
9:   {
10:     case 'C': printf("Casado\n");
11:     case 'S': printf("Solteiro\n");
12:     case 'D': printf("Divorciado\n");
13:     case 'V': printf("Viúvo\n");
14:     default: printf("Estado Civil Incorreto\n");
15:   }
16: }
```

Switch

Observação: Executando o programa gerado pelo código-fonte apresentado no segundo caso, observamos um grave problema. Por exemplo, ao informar o estado civil **S** (solteiro) o resultado impresso será:

Solteiro
Divorciado
Viúvo
Estado Civil Incorreto

Dessa forma, o programa irá executar tudo a partir do **case** correto, incluindo o **default**.

Switch

Nota de Bolso

Sempre que existe coincidência entre a expressão de um **switch** e uma das constantes possíveis para essa expressão são executadas todas as instruções associadas ao **case** correspondente e seguintes, até que o **switch** termine ou seja encontrada a instrução **break** (ou seja encontrada a instrução **return**).

O último **case** ou o **default** de um **switch** não necessita de **break** porque depois de executar as instruções associadas ao último **case** termina a instrução **switch**.

A seguir será apresentado o mesmo exemplo anterior sem resultados inesperados. Observe o exemplo que segue:

Switch

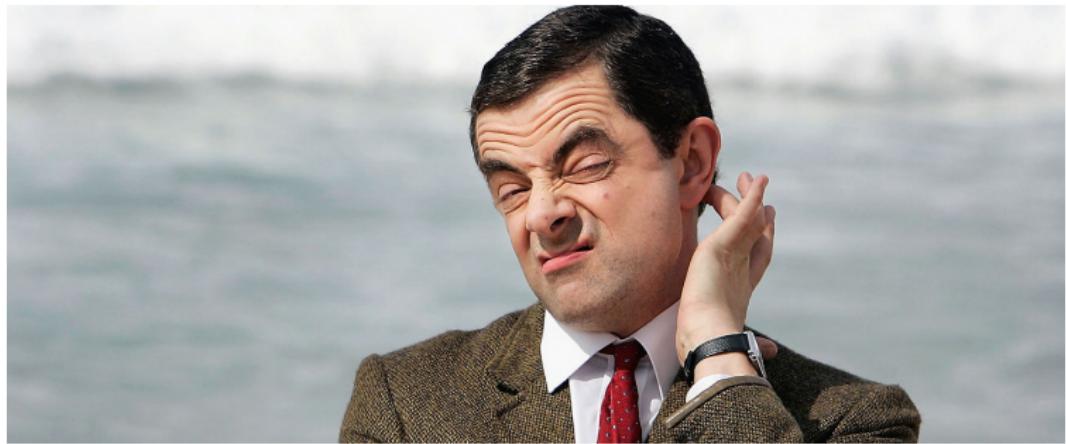
Exemplo Correto:

```
1: #include <stdio.h>
2:
3: void main()
4: { char Est_Civil;
6:   printf("Qual o estado Civil: ");
7:   scanf(" %c", &Est_Civil); /* Ou usar getchar(); */
8:   switch(Est_Civil)
9:   {
10:     case 'C': printf("Casado\n"); break;
11:     case 'S': printf("Solteiro\n"); break;
12:     case 'D': printf("Divorciado\n"); break;
13:     case 'V': printf("Viúvo\n"); break;
14:     default: printf("Estado Civil Incorreto\n");
15:   }
16: }
```

Switch

Poderemos utilizar essa característica de C para construir um programa que funcione igual ao primeiro caso, isto é, o indivíduo poderá informar o estado civil com uma letra maiúscula ou minúscula?

Poderemos utilizar essa característica de C para construir um programa que funcione igual ao primeiro caso, isto é, o indivíduo poderá informar o estado civil com uma letra maiúscula ou minúscula?



Switch

A resposta é SIM. Considere o código abaixo:

```
1: #include <stdio.h>
2:
3: void main()
4: { char Est_Civil;
6:   printf("Qual o estado Civil: ");
7:   scanf(" %c", &Est_Civil); /* Ou usar getchar(); */
8:   switch(Est_Civil)
9:   {
10:     case 'c':
11:     case 'C': printf("Casado\n"); break;
12:     case 's':
13:     case 'S': printf("Solteiro\n"); break;
14:     case 'd':
```

Switch

```
15:     case 'D': printf("Divorciado\n"); break;
16:     case 'v':
17:     case 'V': printf("Viúvo\n"); break;
18:     default: printf("Estado Civil Incorreto\n");
19: }
20:}
```

Loop

Na programação, é preciso aprendermos fazer laço. As instruções de laços também são instruções de controle de fluxo assim como as instruções if, else e switch.

Loop

Na programação, é preciso aprendermos fazer laço. As instruções de laços também são instruções de controle de fluxo assim como as instruções if, else e switch.



While

A instrução `while` executa uma instrução ou bloco de instruções enquanto uma determinada condição permanece verdadeira. Sua sintaxe é:

```
while (condição)
    instrução;
```

Funcionamento da instrução `while`:

While

A instrução `while` executa uma instrução ou bloco de instruções enquanto uma determinada condição permanece verdadeira. Sua sintaxe é:

```
while (condição)
    instrução;
```

Funcionamento da instrução `while`:

- ① A condição é avaliada;

While

A instrução `while` executa uma instrução ou bloco de instruções enquanto uma determinada condição permanece verdadeira. Sua sintaxe é:

```
while (condição)
    instrução;
```

Funcionamento da instrução `while`:

- ① A condição é avaliada;
- ② Se o resultado da avaliação for **Falso** (zero), o laço termina e o programa continua na instrução imediatamente depois do `while`;

While

A instrução `while` executa uma instrução ou bloco de instruções enquanto uma determinada condição permanece verdadeira. Sua sintaxe é:

```
while (condição)
    instrução;
```

Funcionamento da instrução `while`:

- ① A condição é avaliada;
- ② Se o resultado da avaliação for **Falso** (zero), o laço termina e o programa continua na instrução imediatamente depois do `while`;
- ③ Se o resultado da avaliação da condição for **Verdade** (diferente de zero), é executada a instrução (ou bloco de instruções) associado ao `while`;

While

A instrução `while` executa uma instrução ou bloco de instruções enquanto uma determinada condição permanece verdadeira. Sua sintaxe é:

```
while (condição)
    instrução;
```

Funcionamento da instrução `while`:

- ① A condição é avaliada;
- ② Se o resultado da avaliação for **Falso** (zero), o laço termina e o programa continua na instrução imediatamente depois do `while`;
- ③ Se o resultado da avaliação da condição for **Verdade** (diferente de zero), é executada a instrução (ou bloco de instruções) associado ao `while`;
- ④ Volta-ser ao ponto 1.

While

Exercício: Escreva um programa que coloque na tela os primeiros 10 números inteiros.

While

Exercício: Escreva um programa que coloque na tela os primeiros 10 números inteiros.

Qual o erro do programa abaixo?

```
1: #include <stdio.h>
2:
3: void main(void)
4: {
5:     int i;
6:     i=1;
7:     while (i <= 10)
8:         printf("%d\n", i);
9:
10:}
```

While

O programa escrito de forma correta:

```
1: #include <stdio.h>
2:
3: void main(void)
4: {
5:     int i;
6:     i=1;
7:     while (i <= 10)
8:     {
9:         printf("%d\n", i);
10:        i = i+1;
11:    }
12:}
```

While

Pergunta: Qual seria a saída do programa apresentado logo acima caso não fossem colocadas as chaves na instrução do while?

Explique!

Exercício: Resposta da pergunta anterior.

While

Exercício: O que faz o programa abaixo? Escreva sua saída.

```
1: #include <stdio.h>
2:
3: void main(void)
4: {
5:     int n;
6:     n = 10;
7:     while (n)
8:     {
9:         printf("%d\n", n);
10:        n = n-1;
11:    }
12:}
```

While

Exercício: Escreva um programa que coloque na tela a tabuada do número 5.

While

Exercício: Escreva um programa que coloque na tela a tabuada do número 5.

Resposta:

```
1: #include <stdio.h>
2:
3: void main(void)
4: {
5:     int n;
6:     n = 1;
7:     while (n<=10)
8:     {
9:         printf("5 * %2d = %2d\n", n, 5*n);
10:        n = n+1;
11:    }
12: }
```

While

Exercício: Reescreva o programa anterior de modo a apresentar a tabuada de qualquer número introduzido pelo usuário.

While

Resposta:

```
1: #include <stdio.h>
2:
3: void main(void)
4: {
5:     int n, num;
6:     printf(Introduza um N%c: ", 167);
7:     scanf("%d", &num);
8:     n = 1;
9:     while (n<=10)
10:    {
11:        printf("%2d * %2d = %2d\n", num, n, num*n);
12:        n = n+1;
13:    }
14:}
```

While

Exercício: Escreva um programa que imprima em tela as tabuadas de 1 a 10, pulando uma linha entre cada tabuada.

While

Exercício: Escreva um programa que imprima em tela as tabuadas de 1 a 10, pulando uma linha entre cada tabuada.

```
1: #include <stdio.h>
2:
3: void main(void)
4: {
5:     int n, num;
6:
7:     num = 1;
8:     while(num <= 10)
9:     {
10:         n = 1;
11:         while(n <= 10)
12:         {
```

While

```
13:     printf("%2d * %2d = %2d\n", num, n, num * n);
14:     n = n + 1;
15: } /* Indo para a próxima tabuada */
16: num = num + 1;
17: putchar('\n');
18: }
19:}
```

For

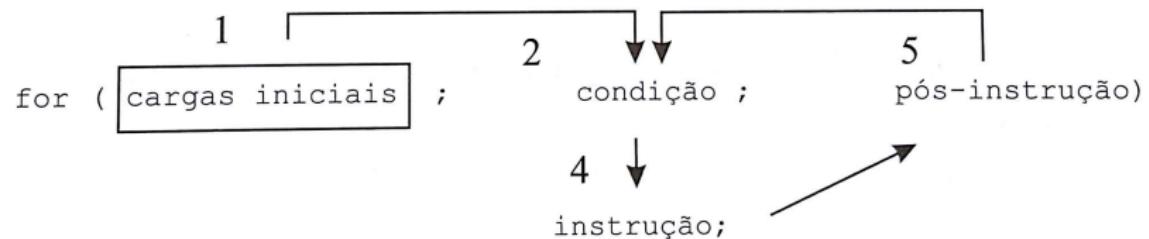
A instrução **for** adapta-se particularmente a situações em que o número de iterações é conhecido a priori. Sua sintaxe é:

```
for (cargas iniciais; condição; pós-instrução)
    instrução;
```

Muito embora possa parecer um laço estranho, o **for** é muito bem desenhada.

Observação: Com a instrução **for** também poderá ser executado um bloco de instruções.

For



O funcionamento do laço **for** pode ser entendido da forma que segue:

- ① O código presente em cargas iniciais é executado. Tal componente é executada apenas uma única vez;
- ② A **condição** é avaliada;
- ③ Se a condição retornar valor **Falso** (zero), o laço termina;
- ④ Se o resultado da condição for Verdade (não zero), então é executada a **instrução**;
- ⑤ Depois de executada a instrução do laço, é executada a **pós-instrução**;
- ⑥ Volta ao ponto 2.

For

Exemplo: Como trocar o comando **while** pelo comando **for** no código abaixo de modo a produzir a mesma saída?

```
1: #include <stdio.h>
2:
3: void main(void)
4: {
5:     int i;
6:     i = 1;
7:     while (i <= 10)
8:     {
9:         printf("%d\n", i);
10:        i = i + 1;
11:    }
12:}
```

For

Exemplo: Como trocar o comando **while** pelo comando **for** no código abaixo de modo a produzir a mesma saída?

```
1: #include <stdio.h>
2:
3: void main(void)
4: {
5:     int i;
6:     i = 1;
7:     for (i = 1; i <= 10; i = i+1)
8:         printf("%d\n", i);
9: }
```

For

Exercício: Escreva um programa que calcule a soma e o produto dos n primeiros números naturais.

Exercício: Escreva um programa que coloque na tela as 10 tabuadas, parando a tela depois de cada uma delas ser escrita. O usuário terá que receber uma mensagem pedindo que seja teclado a tecla **ENTER** para que a próxima tabuada seja calculada e exibida.
(Dicas: Não deverá ser utilizado **while**. Além disso, utilize a função **getchar** para parar a tela)

Importante

Qualquer laço **for** poderá sempre ser reescrito como um laço **while**.

do ... while

A instrução **do ... while** difere das instruções de controle de fluxo anteriores porque o teste da condição é realizado no fim do corpo (instrução ou bloco de instruções) do laço e não antes como ocorre com os laços **while** e **for**.

O que isso acarreta?

Resposta: O corpo do laço (instrução ou bloco de instruções) será executado ao menos uma vez independente da condição retornar **Falso** (zero) ou **Verdade** (número diferente de zero).

do ... while

Sintaxe da instrução do ... while:

```
do  
    instrução;  
while (condição);
```

Exercício: Escreva um programa que apresente um menu com as opções Clientes, Fornecedores, Encomendas e Sair. O programa deve apresentar a opção escolhida pelo usuário até que este deseje sair.

do ... while

```
1: #include <stdio.h>
2:
3: void main(void)
4: {
5:     char opcao;
6:     do
7:     {
8:         printf("\tMENU PRINCIPAL\n");
9:         printf("\n\n\t\tClientes");
10:        printf("\n\n\t\tFornecedores");
11:        printf("\n\n\t\tEncomendas");
12:        printf("\n\n\t\tSair");
13:        printf("\n\n\t\tOpcao: ");
14:        scanf(" %c",&opcao);
15:        fflush(stdin); /* Limpando o Buffer */
```

do ... while

```
16:     switch (opcao)
17:     {
18:         case 'c':
19:             case 'C': puts("Op. CLIENTES"); break;
20:             case 'f':
21:                 case 'F': puts("Op. FORNECEDORES"); break;
22:                 case 'e':
23:                     case 'E': puts("Op. ENCOMENDAS"); break;
24:                     case 's':
25:                         case 'S': break; /* Não faz nada */
26:                         default: puts("Op. INVÁLIDA!!!!");
27:     }
28: }
29: while (opcao != 's' && opcao != 'S');
30: }
```

Instrução **break**

A instrução **break** já é conhecida e utilizamos com o papel de terminar o conjunto das instruções dentro da instrução **switch**. Porém, poderemos utilizar a instrução **break** para terminar um laço (**for**, **while** e **do ... while**). Normalmente a instrução **break** com o intuito de terminar um laço é utilizada dentro de uma instrução **if**.

Instrução break

Exercício: Qual a saída do programa que segue?

```
1: #include <stdio.h>
2:
3: void main(void)
4: {
5:     int i;
6:     for (i=1; i<=100; i=i+1)
7:         if (i == 30)
8:             break;
9:         else
10:             printf("%2d\n", 2*i);
11:     printf("FIM DO LAÇO\n");
12: }
```

Instrução **break**

Exercício: Qual a saída do programa que segue?

```
1: #include <stdio.h>
2:
3: void main(void)
4: {
5:     int i;
6:     for (i=1; i<=100; i=i+1)
7:         if (i == 30)
8:             break;
9:         else
10:             printf("%2d\n", 2*i);
11:     printf("FIM DO LAÇO\n");
12: }
```

Resposta: Irá mostrar os primeiros 29 números pares (2, 4, ..., 58).

Instrução **continue**

A instrução **continue** dentro de um laço permite que a instrução ou bloco de instruções pertencentes ao laço seja terminada passando para a próxima iteração do loop.

Lembre-se

A instrução **continue**, quando presente dentro de um laço, passa o laço para a próxima iteração.

Instrução **continue**

Exercício: Qual a saída do programa que segue?

```
1: #include <stdio.h>
2:
3: void main(void)
4: {
5:     int i;
6:     for (i=1; i<=100; i=i+1)
7:         if (i==60)
8:             break;
9:         else
10:            if (i%2==1) /* se i for ímpar */
11:                continue;
12:            else
13:                printf("%2d\n",i);
14:    printf("FIM DO LAÇO\n");
15:}
```

Outros detalhes

Lembre-se

A instrução **continue** só poderá ser utilizada dentro de loops, enquanto **break** pode ser utilizada em loops ou na instrução **switch**.

Exercício: Escreva um programa que coloque os seguintes números na tela:

1

1 2

1 2 3

...

1 2 3 4 5 6 7 8 9 10

Observação: O programa deverá ser escrito de tal forma que possa ser generalizado para situações de sequências maiores de números.

Outros detalhes

```
1: #include <stdio.h>
2: #define N 10 /* Constante Simbólica */
3:
4: void main(void)
5: {    int i, j;
6:
7:     for (i = 1; i <= N; i = i+1)
8:     {
9:         for (j = 1; j <= N; j = j+1)
10:            {   printf("%d ", j);
11:                if (j == i)
12:                    break;
13:            }
14:            putchar('\n');
15:    }
16:}
```

Alguns exercícios

Exercício: Construa um programa em C que calcule a soma de uma quantidade arbitrária de números reais. O usuário deverá ser perguntado a respeito da quantidade de elementos a ser somados. Com essa informação, utilize uma estrutura de laço para solicitar cada observação a ser somada.

Alguns exercícios

Exercício: Construa um programa em C que calcule a soma de uma quantidade arbitrária de números reais. O usuário deverá ser perguntado a respeito da quantidade de elementos a ser somados. Com essa informação, utilize uma estrutura de laço para solicitar cada observação a ser somada.

Exercício: Construa um programa em C que calcule a variância amostral de uma sequência de observações passadas pelo usuário.

Alguns exercícios

Exercício: Construa um programa em C que calcule a soma de uma quantidade arbitrária de números reais. O usuário deverá ser perguntado a respeito da quantidade de elementos a ser somados. Com essa informação, utilize uma estrutura de laço para solicitar cada observação a ser somada.

Exercício: Construa um programa em C que calcule a variância amostral de uma sequência de observações passadas pelo usuário.

Exercício: Calcule um programa em C que calcule o desvio padrão amostral de uma sequência de observações passadas pelo usuário.

Alguns exercícios

Exercício: Construa um programa em C que calcule a soma de uma quantidade arbitrária de números reais. O usuário deverá ser perguntado a respeito da quantidade de elementos a ser somados. Com essa informação, utilize uma estrutura de laço para solicitar cada observação a ser somada.

Exercício: Construa um programa em C que calcule a variância amostral de uma sequência de observações passadas pelo usuário.

Exercício: Calcule um programa em C que calcule o desvio padrão amostral de uma sequência de observações passadas pelo usuário.
(Dica: utilize a função `sqrt()` da biblioteca **math.h**, isto é, faça `#include <math.h>` para ter acesso à função raiz quadrada).

Outros detalhes

Constante Simbólica

Observe a segunda linha do programa anterior. A instrução `#define N 10` define uma constante simbólica **N** com o valor **10** que será substituído em toda ocorrência de **N** no código fonte na fase de pré-processamento. Além disso é importante saber que constantes simbólicas não existem fisicamente na memória. Trata-se apenas de uma substituição do caractere **N** do código-fonte pelo caractere **10**.

Loops Infinitos

```
while (1)           for ( ; ; )           do
    instrução;       instrução;         instrução;
                      while (1)
```

Operadores ++ e --

Nos códigos em que apresentaram uso de loops era comum incrementarmos ou decrementarmos uma variável ($i = i + 1$ ou $i = i - 1$) o que está perfeitamente correto. Porém, a linguagem C permite-nos incrementar ou decrementar uma variável com uma sintaxe mais enxuta.

Exemplo:

```
1: #include <stdio.h>
2: void main(void)
3: { int i=1;
4:   while (i <= 10)
5:     { printf("%d\n", i);
6:       i = i+1; /* Incrementando variável i */
7:     }
8: }
```

Operadores ++ e --

Nos códigos em que apresentaram uso de loops era comum incrementarmos ou decrementarmos uma variável ($i = i + 1$ ou $i = i - 1$) o que está perfeitamente correto. Porém, a linguagem C permite-nos incrementar ou decrementar uma variável com uma sintaxe mais enxuta.

Exemplo:

```
1: #include <stdio.h>
2: void main(void)
3: { int i=1;
4:   while (i <= 10)
5:     { printf("%d\n", i);
6:       i++; /* Incrementando variável i */
7:     }
8: }
```

Operadores ++ e --

Tabela: Operadores unários de incremento e decremento.

Operador	Significado	Exemplo
++	Incremento de 1	i++ , ++k
--	Decremento de 1	j-- , --alpha

Operadores ++ e --

Tabela: Operadores unários de incremento e decremento.

Operador	Significado	Exemplo
++	Incremento de 1	i++ , ++k
--	Decremento de 1	j-- , --alpha

Tabela: Operadores e suas equivalências.

Operador	Exemplo	Equivalente
++	x++ ou ++x	$x = x + 1$
--	x-- ou --x	$x = x - 1$

Operadores ++ e --

**OK. MAS EXISTE ALGUMA DIFERENÇA ENTRE $++x$ e
 $x++$ ASSIM COMO $--x$ e $x--$?**

Operadores ++ e --

**OK. MAS EXISTE ALGUMA DIFERENÇA ENTRE $++x$ e
 $x++$ ASSIM COMO $--x$ e $x--$?**



Operadores ++ e --

Resposta: Depende. Caso o interesse seja unicamente utilizar os operadores ++ e/ou -- para incrementar ou decrementar uma variável de controle de um loop, tanto faz **incrementar/decrementar** antes ou depois do nome da variável. Já se o interesse é passar o resultado do incremento ou decremento para uma outra variável, o resultado poderá ser diferente. Observe a Tabela que segue para entender:

Operadores ++ e --

Resposta: Depende. Caso o interesse seja unicamente utilizar os operadores ++ e/ou -- para incrementar ou decrementar uma variável de controle de um loop, tanto faz **incrementar/decrementar** antes ou depois do nome da variável. Já se o interesse é passar o resultado do incremento ou decremeno para uma outra variável, o resultado poderá ser diferente. Observe a Tabela que segue para entender:

Tabela: Diferenças entre incrementar antes ou depois da variável.

$y=x++;$	$y=++x;$
Acontece duas coisas: 1. x é atribuído a y 2. x é incrementado	Acontece duas coisas: 1. x é incrementado 2. x é atribuído a y

Operadores ++ e --

Resposta: Depende. Caso o interesse seja unicamente utilizar os operadores ++ e/ou -- para incrementar ou decrementar uma variável de controle de um loop, tanto faz **incrementar/decrementar** antes ou depois do nome da variável. Já se o interesse é passar o resultado do incremento ou decremeno para uma outra variável, o resultado poderá ser diferente. Observe a Tabela que segue para entender:

Tabela: Diferenças entre decrementar antes ou depois da variável.

$y=x--;$	$y=--x;$
Acontece duas coisas: 1. x é atribuído a y 2. x é decrementado	Acontece duas coisas: 1. x é decrementado 2. x é atribuído a y

Operadores ++ e --

Exercício: Implemente o código que segue e explique como o incremento está sendo realizado.

```
1: #include <stdio.h>
2:
3: void main(void)
4: {
5:     int i;
6:     i = 1;
7:     while (i <= 10)
8:         printf("%d\n", i++);
9: }
```

Operadores ++ e --

Exercício: Qual a saída do seguinte programa?

```
1: #include <stdio.h>
2:
3: void main(void)
4: {
5:     int a = 7, b = 7;
6:
7:     printf(" a = %d e b = %d\n", a--, --b);
8:     printf(" a = %d e b = %d\n", a, b);
9: }
```

Operadores ++ e --

Fique Atento!

Jamais tente utilizar o operador ++ ou -- em variáveis que apareçam mais do que uma vez em uma mesma expressão. Evite coisas como `i++ + ++i`.

Lembre-se também que se os operadores ++ ou -- forem utilizados na condição de um **if-else** ou em qualquer outra instrução de controle de fluxo, mesmo que a condição venha a ser **FALSA**, a variável em questão é incrementada ou decrementada.

Exemplo: Abaixo a instrução do **if** não será executada mas **i** assumirá o valor **1** após a instrução **if**.

```
int i = 0;  
if (i++)  
    instrução_do_if;  
printf("%d", i);
```

Atribuição Composta

Em C, é possível reduzir o código utilizando atribuição composta, principalmente em situações em que as variáveis apresentam nomes bastante longos.

Exemplo: Considere os trechos de códigos que seguem, onde temos uma variável e queremos incrementar esta com o valor 7. Temos duas formas de fazer isso:

- ① `minhaLindaVariavelQueAdoro =
 minhaLindaVariavelQueAdoro + 7;`
- ② `minhaLindaVariavelQueAdoro += 7;`

Nota: Essas duas formas de atribuição são equivalentes e o uso da segunda forma é comum mesmo em situações em que o nome da variável seja curto.

Atribuição Composta

A Tabela que segue resume a ideia de atribuição composta para outros operadores e apresenta o significado dessas atribuições.

Tabela: Exemplos e significados de atribuição composta para diversos operadores em C.

Exemplo	Significado
$x += 1$	$x = x + 1$
$y *= 2+3$	$y = y * (2 + 3)$
$a -= b + 1$	$a = a - (b + 1)$
$k /= 12$	$k = k / 12$
$r %= 2$	$r = r \% 2$

Observação: O operador utilizado na atribuição composta deverá estar junto com o operador de atribuição $=$.

Funções

A melhor forma de se desenvolver programas em C é dividir o problema em partes menores que fazem uma tarefa bem específica.

Muito embora, no momento presente, não sabemos construir funções, a todo momento em nossos códigos fizemos uso de diversas funções em C. Por exemplo, utilizamos as funções **printf**, **scanf**, **getchar**, **putchar**, **fflush**, entre outras.

Observação

Uma vez que temos uma função bem implementada e o mais geral possível, não precisamos implementar esta função novamente caso necessitamos utilizar em novos códigos C. No popular, não é preciso reinventar a roda.

Nota

Muitas das funções que utilizamos já são funções implementadas em alguma biblioteca C. Sendo assim, antes de implementar uma função é razoável fazer uma pesquisa (Web, livros e/ou tutoriais) por bibliotecas C. Muitas vezes não precisamos implementar alguma tarefa visto que elas estão implementadas em alguma biblioteca.

Porém, não é difícil nos depararmos com problemas em que necessitamos de uma função que não esteja implementada em nenhuma biblioteca. Esse fato, na verdade, é bastante comum nas ciências exatas. Assim,

Funções

Nota

Muitas das funções que utilizamos já são implementadas em alguma biblioteca C. Sendo assim, antes de implementar uma função é razoável fazer uma pesquisa (Web, livros e/ou tutoriais) por bibliotecas C. Muitas vezes não precisamos implementar alguma tarefa visto que elas estão implementadas em alguma biblioteca.

Porém, não é difícil nos depararmos com problemas em que necessitamos de uma função que não esteja implementada em nenhuma biblioteca. Esse fato, na verdade, é bastante comum nas ciências exatas. Assim,

PRECISAMOS IMPLEMENTAR NOSSAS FUNÇÕES

Funções

Dicas Úteis:

Funções

Dicas Úteis:

- ① Familiarize-se com o ótimo conjunto de funções da biblioteca padrão ANSI C.

Funções

Dicas Úteis:

- ① Familiarize-se com o ótimo conjunto de funções da biblioteca padrão ANSI C.
- ② Evite reinventar a roda. **Quando possível**, use as funções da biblioteca padrão do ANSI C em vez de escrever novas funções. Isso reduz o tempo de desenvolvimento de programas.

Funções

Dicas Úteis:

- ① Familiarize-se com o ótimo conjunto de funções da biblioteca padrão ANSI C.
- ② Evite reinventar a roda. **Quando possível**, use as funções da biblioteca padrão do ANSI C em vez de escrever novas funções. Isso reduz o tempo de desenvolvimento de programas.
- ③ Utilizar as funções da biblioteca padrão ANSI C ajuda a tornar os programas mais portáveis.

Funções da Biblioteca Matemática

As funções da biblioteca matemática permitem ao programador realizar determinados cálculos matemáticos comuns. Usamos várias funções matemáticas aqui para introduzir o conceito de funções.

Funções da Biblioteca Matemática

As funções da biblioteca matemática permitem ao programador realizar determinados cálculos matemáticos comuns. Usamos várias funções matemáticas aqui para introduzir o conceito de funções.

Sintaxe de uso de funções:

```
nome_da_função(parâmetros_da_função);
```

Funções da Biblioteca Matemática

As funções da biblioteca matemática permitem ao programador realizar determinados cálculos matemáticos comuns. Usamos várias funções matemáticas aqui para introduzir o conceito de funções.

Sintaxe de uso de funções:

```
nome_da_função(parâmetros_da_função);
```

Observação: Se a função possuir mais de um parâmetro, estes serão separados por vírgula.

Funções da Biblioteca Matemática

As funções da biblioteca matemática permitem ao programador realizar determinados cálculos matemáticos comuns. Usamos várias funções matemáticas aqui para introduzir o conceito de funções.

Sintaxe de uso de funções:

```
nome_da_função(parâmetros_da_função);
```

Observação: Se a função possuir mais de um parâmetro, estes serão separados por vírgula.

Exemplo:

```
printf("%.2f", sqrt(900.0));
```

Muito Importante

Algo muito importante foi mostrado no exemplo anterior. Note que **sqrt** é uma função que recebe um argumento que, no caso do exemplo, a função **sqrt** recebe um argumento do tipo **double** 900.0. Porém, a **printf** também é uma função que recebe como argumento a função **sqrt** avaliada no parâmetro **double** 900.0.

Muito embora **sqrt(900.0)** é um número, mais a frente iremos ver como passar funções como argumento para outra função. No caso do exemplo, a função **printf** receberá o resultado de **sqrt(900.0)** que é do tipo **double** e não a função **sqrt** propriamente dita.

Funções da Biblioteca Matemática

Nota

Na verdade, a ideia de chamar funções dentro de funções sempre foi uma realidade nos nossos programas anteriores. Lembre-se que a **função principal** em que todo programa C deverá possuir é a função **main**. Ao utilizar **printf**, **scanf**, **getchar**, entre outras funções, a chamada à essas funções foram feitas dentro da função **main**.

Dessa forma, poderemos criar uma nova função que faz chamada a outras funções que já implementamos anteriormente ou mesmo que estejam implementadas em alguma biblioteca.

Funções da Biblioteca Matemática

Teremos acessos à funções de bibliotecas externas informando por meio de diretivas de pré-processador os arquivos cabeçalho (header) das funções das bibliotecas utilizando a diretiva `#include <biblioteca.h>`.

Observações:

Funções da Biblioteca Matemática

Teremos acessos à funções de bibliotecas externas informando por meio de diretivas de pré-processador os arquivos cabeçalho (header) das funções das bibliotecas utilizando a diretiva `#include <biblioteca.h>`.

Observações:

- ① Inclua o arquivo de cabeçalho (header) matemático usando a diretiva do pré-processador `#include <math.h>` para ter acesso à funções matemáticas básicas.

Funções da Biblioteca Matemática

Teremos acessos à funções de bibliotecas externas informando por meio de diretivas de pré-processador os arquivos cabeçalho (header) das funções das bibliotecas utilizando a diretiva `#include <biblioteca.h>`.

Observações:

- ① Inclua o arquivo de cabeçalho (header) matemático usando a diretiva do pré-processador `#include <math.h>` para ter acesso à funções matemáticas básicas.
- ② Esquecer de incluir o arquivo de cabeçalho é um erro comum dos programadores. Na maioria dos casos será obtido um erro de compilação ou mesmo resultados inesperados.

Funções da Biblioteca Matemática

Tabela: Algumas funções matemáticas da biblioteca matemática ANSI C.

Função	Descrição	Exemplo
<code>sqrt(x)</code>	raiz quadrada de x	<code>sqrt(900.0)</code> é 30.0
<code>exp(x)</code>	função exponencial e^x	<code>exp(1.0)</code> é 2.718282
<code>log(x)</code>	logaritmo natural de x	<code>log(2.7182818)</code> é 1.0
<code>log10(x)</code>	logaritmo de x (base 10)	<code>log10(10.0)</code> é 1.0
<code>fabs(x)</code>	valor absoluto de x	<code>fabs(-.1)</code> é 0.1
<code>pow(x,y)</code>	x elevado à potência y x^y	<code>pow(2,7)</code> é 128.0
<code>sin(x)</code>	seno de x (em radianos)	<code>sin(0.0)</code> é 0.0
<code>cos(x)</code>	cosseno de x (em radianos)	<code>cos(0.0)</code> é 1.0
<code>tan(x)</code>	tangente de x (em radianos)	<code>tan(0.0)</code> é 0.0
<code>ceil(x)</code>	arredonda para o maior inteiro	<code>ceil(9.2)</code> é 10.0
<code>floor(x)</code>	arredonda para o menor inteiro	<code>floor(9.2)</code> é 9.0

Protótipo de Função

Exemplo: Vamos criar uma função chamada **square** que eleva ao quadrado um número do tipo **int**.

```
1: #include <stdio.h>
2:
3: int square(int); /* prototipo da função */
4:
5: int main(void) /* poderíamos utilizar main(void) */
6: {
7:     int x;
8:
9:     for (x = 1; x <=10; x++)
10:         printf("%d", square(x));
11:     printf("\n");
12:     return 0;
13: }
```

Funções da Biblioteca Matemática

```
14: /* Definição da função */  
15:  
16: int square(int y)  
17: {  
18:     return y * y;  
19: }
```

Note que a função **square** é invocada dentro da função **main**. Além disso perceba que na linha 12 colocamos `return 0` para indicar que o programa foi executado corretamente. Uma vez que retornamos **0 (inteiro)** precisamos que a função **main** retorne o tipo **int**. **Quando não declaramos o tipo de retorno de uma função, o tipo dessa função será int por padrão.** Porém, alguns compiladores poderão exigir que o tipo seja explicitado.

Protótipo de Função

Ainda no exemplo anterior, note a linha 3. Nesta linha é escrito o **protótipo** da função **square**. Sem essa linha iríamos obter um erro de compilação uma vez que a função **square** é chamada dentro da função **main** antes de sua definição que se dá entre as linhas 16 e 19.

Observação

O **protótipo** de uma função informa ao compilador de C da existência de uma função, o tipo de retorno e a quantidade de parâmetros. Na verdade, o compilador não se importa com a quantidade de parâmetros, sendo não necessário sua declaração. Colocamos os tipos dos parâmetros pelo fato de seguir uma boa prática de programação. Tente trocar a linha 3 por `int square();`.

Protótipo de Função

Observação

Uma vez que o compilador conhece o protótipo da função, este não irá emitir uma mensagem de erro quando dentro da função **main** a função **square** for chamada, mesmo que esta função tenha sido implementada depois da função **main**.

Formas que poderíamos considerar para declarar o protótipo da função square:

Protótipo de Função

Observação

Uma vez que o compilador conhece o protótipo da função, este não irá emitir uma mensagem de erro quando dentro da função **main** a função **square** for chamada, mesmo que esta função tenha sido implementada depois da função **main**.

Formas que poderíamos considerar para declarar o protótipo da função square:

- ① int square();

Protótipo de Função

Observação

Uma vez que o compilador conhece o protótipo da função, este não irá emitir uma mensagem de erro quando dentro da função **main** a função **square** for chamada, mesmo que esta função tenha sido implementada depois da função **main**.

Formas que poderíamos considerar para declarar o protótipo da função square:

- ① int square();
- ② int square(int);

Protótipo de Função

Observação

Uma vez que o compilador conhece o protótipo da função, este não irá emitir uma mensagem de erro quando dentro da função **main** a função **square** for chamada, mesmo que esta função tenha sido implementada depois da função **main**.

Formas que poderíamos considerar para declarar o protótipo da função square:

- ① int square();
- ② int square(int);
- ③ int square(int NomeVariavel);. Por exemplo, int square(int numero);

Protótipo de Função

A terceira forma serve apenas para ajudar ao programador. Note também que utilizando a terceira forma, o nome da variável que no caso do exemplo é parâmetro da função **square** não tem nenhuma necessidade de ser igual ao nome da variável que será passada à função **square** dentro da função **main**.

Protótipo de Função

- ① Omitir o tipo do valor de retorno em uma definição de uma função causa erro de sintaxe se o protótipo da função especificar um tipo de retorno diferente de **int**. Isso porque omitir o tipo de retorno de uma função faz com que o compilador assuma que o tipo retornado pela função é do tipo **inteiro**. Porém, alguns compiladores poderão exigir que o tipo de uma função seja especificado.

Protótipo de Função

- ① Omitir o tipo do valor de retorno em uma definição de uma função causa erro de sintaxe se o protótipo da função especificar um tipo de retorno diferente de **int**. Isso porque omitir o tipo de retorno de uma função faz com que o compilador assuma que o tipo retornado pela função é do tipo **inteiro**. Porém, alguns compiladores poderão exigir que o tipo de uma função seja especificado.
- ② Uma função do tipo **void** jamais poderá retornar nenhum valor, isto é, não poderá ter a instrução **return**.

Protótipo de Função

- ① Omitir o tipo do valor de retorno em uma definição de uma função causa erro de sintaxe se o protótipo da função especificar um tipo de retorno diferente de **int**. Isso porque omitir o tipo de retorno de uma função faz com que o compilador assuma que o tipo retornado pela função é do tipo **inteiro**. Porém, alguns compiladores poderão exigir que o tipo de uma função seja especificado.
- ② Uma função do tipo **void** jamais poderá retornar nenhum valor, isto é, não poderá ter a instrução **return**.
- ③ Nunca esqueça de colocar um **ponto-e-vírgula** ao final da declaração do protótipo de uma função. Protótipos de uma função são instruções C.

Protótipo de Função

Forma geral da especificação de um protótipo de uma função:

```
tipo_de_retorno_da_função (lista_de_tipos_de_parâmetros);
```

Observação: Listar os tipos de parâmetros não é necessário.
Porém, se assim escolher, os tipos devem ser separados por vírgula.

Função

Forma geral da especificação de uma função:

```
tipo_de_retorno nome_da_função(lista_de_parâmetros)
{
    /* Bloco de Instruções */

    declarações de variáveis locais;

    instruções da função;

    return instrução; /* Caso se tenha valor
                        a retornar */
}
```

Dentro dos arquivos com extensão **.h** estão os protótipos de funções. Por exemplo, dentro do arquivo **stdio.h** estão os protótipos das funções **printf**, **scanf**, entre outras funções. Já no arquivo **math.h** encontram-se os protótipos de funções das funções matemáticas apresentadas anteriormente.

Exemplo: Voltemos ao código anterior onde criamos a função **square**. Uma vez que queremos aproveitar o código da função **square** em outros programas, poderíamos passar a função **square** e o protótipo desta função para um arquivo com extensão **.h**. Como fazer isso?

Files Heard

Files Heards

- ① Primeiramente coloque a o **protótipo** da função **square** no arquivo **meucabecalho.h** que será criado no mesmo diretório que contem o arquivo **.c**.

Files Heards

- ① Primeiramente coloque o **protótipo** da função **square** no arquivo **meucabecalho.h** que será criado no mesmo diretório que contém o arquivo **.c**.
- ② Depois de especificar o protótipo da função, coloque o código da função **square** também no arquivo

- ① Primeiramente coloque o **protótipo** da função **square** no arquivo **meucabecalho.h** que será criado no mesmo diretório que contém o arquivo **.c**.
- ② Depois de especificar o protótipo da função, coloque o código da função **square** também no arquivo **meucabecalho.h** criado no passo anterior.

Files Heards

- ① Primeiramente coloque o **protótipo** da função **square** no arquivo **meucabecalho.h** que será criado no mesmo diretório que contém o arquivo **.c**.
- ② Depois de especificar o protótipo da função, coloque o código da função **square** também no arquivo **meucabecalho.h** criado no passo anterior.
- ③ No arquivo com extensão **.c** coloque a diretiva de pré-processador **#include "meucabecalho.h"**.

Files Heards

- ① Primeiramente coloque o **protótipo** da função **square** no arquivo **meucabecalho.h** que será criado no mesmo diretório que contém o arquivo **.c**.
- ② Depois de especificar o protótipo da função, coloque o código da função **square** também no arquivo **meucabecalho.h** criado no passo anterior.
- ③ No arquivo com extensão **.c** coloque a diretiva de pré-processador **#include "meucabecalho.h"**.
- ④ Compile o programa **.c**.

Arquivo meucabecalho.h

```
1: /* O nome do arquivo é meucabecalho.h */
2:
3: int square(); /* prototipo da funcao */
4:
5: int square(int y) /* função square */
6: {
7:     return y * y;
8: }
```

Arquivo `codigo.c`

```
1: #include <stdio.h>
2: #include "meucabecalho.h"
3:
4: int main(void)
5: {
6:     int x;
7:
8:     for (x = 1; x<=10; x++)
9:         printf("%d ", square(x));
10:    printf("\n");
11:    return 0;
12:}
```

Arquivo **codigo.c**

```
1: #include <stdio.h>
2: #include "meucabecalho.h"
3:
4: int main(void)
5: {
6:     int x;
7:
8:     for (x = 1; x<=10; x++)
9:         printf("%d ", square(x));
10:    printf("\n");
11:    return 0;
12:}
```

Compile o arquivo `codigo.c` acima.

Lembre-se

A grande vantagem de criar nossos próprios arquivos cabeçalhos (arquivos com extensão **.h**) é que em um outros código que precisarmos de uma função que já implementamos e definimos dentro do nosso cabeçalho, bastará importar o cabeçalho para que nosso novo código tenha acesso as funções implementadas.

Funções

Exercício: Escreva um código C que imprima em tela a saída que segue:

```
*****
Números entre 1 e 5
*****
1
2
3
4
5
*****
```

Detalhes a considerar: Crie uma função para imprimir as linhas formadas pelo símbolo * (**20 caracteres**). Não será necessário criar um arquivo cabeçalho. Coloque o protótipo da função criada bem como a implementação da função no arquivo **.c**.

Funções

Solução

```
1: #include <stdio.h>
2:
3: void linha()
4: {
5:     int i;
6:     for (i=1; i<=20; i++)
7:         putchar('*');
8:     putchar('\n');
9: }
10:
11: int main(void)
12: {
13:     int i;
```

Funções

```
14:     linha();
15:     puts("Números entre 1 e 5");
16:     linha();
17:
18:     for(i=1; i<5; i++)
19:         printf("%d\n", i);
20:
21:     linha();
22:
23:     return 0;
24: }
```

Funções

Exercício: Escreva um código C que imprima em tela a saída que segue:

```
*****
Números entre 1 e 5
*****
1
2
3
4
5
*****
```

Detalhes a considerar: Torne a função **linha** mais geral. Permita ela possa escrever qualquer quantidade do símbolo * e não precisamente 20 asteriscos.

Funções

Solução

```
1: #include <stdio.h>
2:
3: void linha(int num)
4: {
5:     int i;
6:     for (i=1; i<=num; i++)
7:         putchar('*');
8:     putchar('\n');
9: }
10:
11: int main(void)
12: {
13:     int i;
```

Funções

```
14:     linha(20);
15:     puts("Números entre 1 e 5");
16:     linha(20);
17:
18:     for(i=1; i<5; i++)
19:         printf("%d\n", i);
20:
21:     linha(20);
22:
23:     return 0;
24: }
```

Funções

Exercício: Escreva um código C que imprima em tela a saída que segue:

```
*****
Números entre 1 e 5
*****
1
2
3
4
5
*****
```

Detalhes a considerar: Torne a função **linha** ainda mais geral. Permita ela possa escrever qualquer símbolo em qualquer quantidade e não precisamente asteriscos.

Funções

Solução

```
1: #include <stdio.h>
2:
3: void linha(int num, char ch)
4: {
5:     int i;
6:     for (i=1; i<=num; i++)
7:         putchar(ch);
8:     putchar('\n');
9: }
10:
11: int main(void)
12: {
13:     int i;
```

Funções

```
14:     linha(20, '*');
15:     puts("Números entre 1 e 5");
16:     linha(20, '*');
17:
18:     for(i=1; i<5; i++)
19:         printf("%d\n", i);
20:
21:     linha(20, '*');
22:
23:     return 0;
24: }
```

Funções

Qual a diferença entre funções e procedimentos?

Ao contrário de uma **função**, um **procedimento** não devolve nenhum valor. Por exemplo, no último exercício, implementamos a função **linha** que possui dois argumentos, sendo eles os argumentos **num** e **ch**. Porém, observe que **linha** não retorna nenhum valor (isto é, não possui a instrução **return** em sua implementação retornando um valor diferente de **void**). Observe que a função retorna o tipo **void**.

Variáveis Locais

Nota muito importante

Observe que no exemplo anterior a função linha declara uma variável **i** do tipo **int**. Observe também que dentro da função principal **main** é criada uma outra variável **i** também do tipo **int**. Não há problema algum, uma vez que essas variáveis não possuem relação alguma. As variáveis dentro de uma função são **variáveis locais** e são destruídas no término da função.

Na verdade, toda e qualquer variável declarada no interior de qualquer bloco, é **variável local** à este bloco e não é reconhecida pelo bloco mais externo. Por exemplo, se declaramos duas variáveis de mesmo nome, uma no bloco mais externo e a outra no bloco mais interno a referência ao nome da variável no bloco mais interno irá considerar a variável declarada neste bloco, isto é, à variável local.

Variáveis Locais

Exemplo: Para entendermos melhor, implemente o código abaixo.
Explique a saída!

```
1: #include <stdio.h>
2:
3: void main(void)
4: {
5:     float i = 7.0;
6:
7:     {
8:         float i = 77.0;
9:         printf("O valor de i é %f\n", i);
10:    }
11:
12:    printf("O valor de i é %f\n", i);
13:}
```

Variáveis Locais

Exemplo: Observe que nesse exemplo, a variável **j** foi reconhecida no bloco mais interno, uma vez que neste boco não foi definida nenhuma variável **j**.

```
1: #include <stdio.h>
2: void main(void)
3: {    float i = 7.0, j = 9.0;
4:
5:     { /* Início do bloco mais interno */
6:         float i = 77.0;
7:         printf("i = %f e j = %f\n", i, j);
8:     } /* Fim do bloco mais interno */
9:
10:    printf("O valor de i é %f\n", i);
11:}
```

Variáveis Locais

Lembre-se

Variáveis locais são destruídas quando o **bloco** em que estas foram declaradas alcança o fim ou quando a **função** em que elas foram declaradas termina de ser executada.

Ainda sobre funções

Nota

Uma função poderá ter mais de uma instrução **return**. Porém, apenas uma das instruções será executada. Dessa forma, uma função poderá ao final de sua execução devolver apenas um retorno.

Exemplo: Considere a função **max** que retorna o valor máximo de dois números.

```
1: float max(float n1, float n2)
2: {
3:     if (n1 > n2)
4:         return n1;
5:     else
6:         return n2;
7: }
```

Ou será devolvido **return n1** ou **return n2**.

Funções Recursivas

O que é uma função recursiva?

Função recursiva é uma função que chama a si mesma.

Observação: É necessário muita prática do programador com funções recursivas para que o processo de recursividade pareça natural. Iremos introduzir funções recursivas com dois exemplos.

Exemplo: Vamos implementar a função factorial de um número inteiro não negativo. Todos nós sabemos que se n é um número inteiro não negativo, o factorial de n é definido por:

$$n! = n \times (n - 1) \times (n - 2) \times \cdots \times 1,$$

em que $0! = 1$.

Funções Recursivas

Antes de apresentar o exemplo utilizando recursividade, faça o exercício abaixo:

Exercício: Implemente uma função chamada **fatorial** com o que você já sabe de C.

Funções Recursivas

Solução:

```
1: #include <stdio.h>
2:
3: long int fatorial() /* Poderia substituir por
3:                      long int fatorial(long int); */;
4: void main(void)
5: {
6:     long int n;
7:     printf("Entre com o fatorial de um número: ");
8:     scanf("%ld", &n);
9:     printf("O fatorial %ld é %ld\n", n, fatorial(n));
10:}
```

Funções Recursivas

```
11: long int factorial (long int n)
12: {
13:     long int resultado = 1;
14:
15:     if (n == 0) return 1;
16:
17:     while(n!=0)
18:     {
19:         resultado = resultado*n;
20:         n--;
21:     }
22:
23:     return resultado;
24: }
```

Funções Recursivas

Uma observação da função **fatorial** implementada no exercício acima é que certamente você irá ter problemas para calcular fatorial de números grandes. Por exemplo, tente fazer **fatorial(100)**.

Mas por quê isto ocorre?

O número $100!$ é 9.332622×10^{157} . Em um computador de 64 bits não se terá bits suficientes para representar este número.

Funções Recursivas

Mas professor, não tem jeito de trabalhar com números maiores? Eu vi que no software Mathematica eu consigo calcular fatoriais muito maiores que este sem grandes problemas

Mas professor, não tem jeito de trabalhar com números maiores? Eu vi que no software Mathematica eu consigo calcular fatoriais muito maiores que este sem grandes problemas

Resposta:

Tem como! Basta utilizar bibliotecas **bibliotecas para aritmética de precisão arbitrária** em que não há limite prático para a precisão, exceto os implicados pela memória disponível na máquina. Uma boa biblioteca é a **GMP (GNU Multiple Precision)**.

Detalhes sobre a **GNU Multiple Precision** poderá ser obtido em
<https://gmplib.org/>.

Funções Recursivas

O código abaixo é apenas para os curiosos. Para que funcione é preciso baixar a biblioteca **GNU Multiple Precision** e fazer com que a IDE de programação (no caso o Code::Blocks) seja configurado para reconhecer a biblioteca. Para dominar entender a biblioteca, leia a documentação disponível no site.

```
#include <stdio.h>
#include <gmp.h>
void fatorial();
void main(void)
{
    unsigned long int n;
    mpz_t saida;
    mpz_init(saida);
    printf("Entre com o fatorial de um número: ");
    scanf("%ld", &n); fatorial(saida, n);
    mpz_out_str(stdout, 10, saida); putchar('\n');}
```

Funções Recursivas

```
void fatorial (mpz_t saida, unsigned long int n)
{
    mpz_t result;
    mpz_t N;
    mpz_init(N);
    mpz_init(result);
    mpz_set_d(result, 1);

    while(n!=0)
    {
        mpz_set_d(N, n);
        mpz_mul(result,result, N);
        n--;
    }
    mpz_set(saida, result);
}
```

Funções Recursivas

Voltando ao exemplo anterior, vamos implementar o $n!$ usando recursividade. O código segue abaixo:

```
1: #include <stdio.h>
2:
3: long int fatorial(); /* Poderia substituir por
4:                      long int fatorial(long int); */
5: void main(void)
6: {
7:     long int n;
8:     printf("Introduza o número: ");
9:     scanf("%ld", &n);
10:    printf("O fatorial de %d é: %ld\n", n, fatorial(n));
11: }
```

Funções Recursivas

```
12:     long int fatorial (long int n)
13:     {
14:         if(n < 1) return 1;
15:         else return (n * fatorial(n-1));
16:     }
```

Funções Recursivas

Exemplo: Na matemática, a sucessão de Fibonacci (sequência de Fibonacci) é uma sequência de números inteiros que tem os dois primeiros termos da sequência os números 0 e 1 e cada termo subsequente é obtido somando os dois inteiros anteriores da sequência. A sequência é:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, ...

Construa uma função (**fibonacci**) em C que calcule o número de Fibonacci em uma dada posição da sequência. Por exemplo, se informado **fibonacci(9)** o programa retorne o valor **21**.

Funções Recursivas

```
1: #include <stdio.h>
2:
3: long int fibonacci();
4:
5: void main(void)
6: {
7:     long int n;
8:
9:     printf("Entre com um inteiro: ");
10:    scanf("%ld", &n);
11:    printf("Fibonacci(%ld) = %ld\n", n, fibonacci(n));
12:}
```

Funções Recursivas

```
13: long int fibonacci(long n)
14: {
15:     if (n == 0 || n == 1)
16:         return n;
17:     else
18:         return fibonacci(n-1) + fibonacci(n-2);
19: }
```

Escopo de Variáveis em C

Todo programador de C ou mesmo de outras linguagens de programação deverá entender muito bem o escopo de variáveis. Tais regras deixam o programar ciente em quais partes de um programa C as variáveis declaradas serão reconhecidas. Em C é possível que variáveis **distintas** possuam o mesmo nome.

Exemplo:

```
1: #include <stdio.h>
2:
3: void main(void)
4: {
5:     int i = 7;
6:
7:     { /* Aqui inicia o bloco interno */
8:         int i = 17;
```

Escopo de Variáveis em C

```
9:     printf("No bloco interno, i = %d\n", i);
10: } /* Aqui termina o bloco interno */
11:
11: printf("No bloco externo, i = %d\n", i);
12:}
```

Muito Importante

Na execução do código executável gerado pela compilação do código fonte do exemplo acima podemos perceber que a variável **i** assume diferentes valores. Ou melhor, **na verdade não são as mesmas variáveis muito embora os nomes são idênticos.**

Escopo de Variáveis em C

Muito Importante

Dessa forma, temos que uma variável em um bloco mais interno sempre será local à este bloco. Ou seja, tal variável será destruída ao final da execução deste bloco. Agora compile o código abaixo e execute o programa gerado.

```
1: #include <stdio.h>
2: void main(void)
3: { int i = 7;
4:   { /* Aqui inicia o bloco mais interno */
5:     printf("No bloco interno, i = %d\n", i);
6:   } /* Aqui termina o bloco mais interno */
7: }
```

Escopo de Variáveis em C

Observação: Perceba que uma variável declarada em um bloco mais externo tem escopo no bloco mais interno se neste bloco não há uma declaração de uma variável local à este bloco.

A linguagem C fornece quatro classes de armazenamento utilizando especificadores no momento de declarar estas variáveis. Esses especificadores são:

Escopo de Variáveis em C

Observação: Perceba que uma variável declarada em um bloco mais externo tem escopo no bloco mais interno se neste bloco não há uma declaração de uma variável local à este bloco.

A linguagem C fornece quatro classes de armazenamento utilizando especificadores no momento de declarar estas variáveis. Esses especificadores são:

① **auto;**

Escopo de Variáveis em C

Observação: Perceba que uma variável declarada em um bloco mais externo tem escopo no bloco mais interno se neste bloco não há uma declaração de uma variável local à este bloco.

A linguagem C fornece quatro classes de armazenamento utilizando especificadores no momento de declarar estas variáveis. Esses especificadores são:

- ① **auto;**
- ② **register;**

Escopo de Variáveis em C

Observação: Perceba que uma variável declarada em um bloco mais externo tem escopo no bloco mais interno se neste bloco não há uma declaração de uma variável local à este bloco.

A linguagem C fornece quatro classes de armazenamento utilizando especificadores no momento de declarar estas variáveis. Esses especificadores são:

- ① **auto;**
- ② **register;**
- ③ **extern;**

Escopo de Variáveis em C

Observação: Perceba que uma variável declarada em um bloco mais externo tem escopo no bloco mais interno se neste bloco não há uma declaração de uma variável local à este bloco.

A linguagem C fornece quatro classes de armazenamento utilizando especificadores no momento de declarar estas variáveis. Esses especificadores são:

- ① **auto;**
- ② **register;**
- ③ **extern;**
- ④ **static.**

Escopo de Variáveis em C

Classe de armazenamento?

Uma **classe de armazenamento** de um **identificador** (nome de uma variável) tem como objetivo determinar seu **tempo de armazenamento, escopo e linkage**.

Um **tempo de armazenamento** de um identificador é o período durante o qual aquele identificador (variável) permanece na memória. Alguns identificadores permanecem pouco tempo, alguns são criados e eliminados repetidamente, e outros permanecem durante toda a execução do programa.

Escopo de Variáveis em C

Classe de armazenamento?

O **escopo** de um identificador (variável) é onde se pode fazer referência àquele identificador dentro de um programa. Pode-se fazer referência a alguns identificadores ao longo de todo o programa, outros apenas a partir de determinados locais de um programa.

A **likage** de um identificador determina, para um programa com vários arquivos-fonte, se um identificador (variável) é conhecido apenas no arquivo-fonte atual ou em qualquer arquivo-fonte com as declarações adequadas. Isso será visto em outro momento.

Escopo de Variáveis em C

As quatro classes de armazenamento dos identificadores podem ser divididas em dois tempos de armazenamento. Para identificar cada classe de armazenamento de uma variável utilizamos palavras-chaves apresentadas anteriormente (**auto**, **register**, **extern**, **static**).

Escopo de Variáveis em C

As quatro classes de armazenamento dos identificadores podem ser divididas em dois tempos de armazenamento. Para identificar cada classe de armazenamento de uma variável utilizamos palavras-chaves apresentadas anteriormente (**auto**, **register**, **extern**, **static**).

① Tempo de armazenamento automático:

Escopo de Variáveis em C

As quatro classes de armazenamento dos identificadores podem ser divididas em dois tempos de armazenamento. Para identificar cada classe de armazenamento de uma variável utilizamos palavras-chaves apresentadas anteriormente (**auto**, **register**, **extern**, **static**).

① Tempo de armazenamento automático:

- ① Palavra chave **auto**;

Escopo de Variáveis em C

As quatro classes de armazenamento dos identificadores podem ser divididas em dois tempos de armazenamento. Para identificar cada classe de armazenamento de uma variável utilizamos palavras-chaves apresentadas anteriormente (**auto**, **register**, **extern**, **static**).

① Tempo de armazenamento automático:

- ① Palavra chave **auto**;
- ② Palavra chave **register**.

② Tempo de armazenamento estático:

Escopo de Variáveis em C

As quatro classes de armazenamento dos identificadores podem ser divididas em dois tempos de armazenamento. Para identificar cada classe de armazenamento de uma variável utilizamos palavras-chaves apresentadas anteriormente (**auto**, **register**, **extern**, **static**).

① Tempo de armazenamento automático:

- ① Palavra chave **auto**;
- ② Palavra chave **register**.

② Tempo de armazenamento estático:

- ① Palavra chave **extern**;

Escopo de Variáveis em C

As quatro classes de armazenamento dos identificadores podem ser divididas em dois tempos de armazenamento. Para identificar cada classe de armazenamento de uma variável utilizamos palavras-chaves apresentadas anteriormente (**auto**, **register**, **extern**, **static**).

① Tempo de armazenamento automático:

- ① Palavra chave **auto**;
- ② Palavra chave **register**.

② Tempo de armazenamento estático:

- ① Palavra chave **extern**;
- ② Palavra chave **static**.

Atenção

As **variáveis locais** possuem tempo de armazenamento automático por default. Portanto, a palavra chave **auto** raramente é usada.

É sempre importante utilizar a teoria do privilégio mínimo que consiste em não conceder privilégio de modificações de variáveis utilizadas em diversos lugares de um programa à funções caso estas não tenham de fato a necessidade de alterar variáveis. Isso evitaria efeitos colaterais indesejáveis caso não fosse intenção do programador modificar referencialmente (na memória) algumas variáveis.

Além disso, porquê armazenar variáveis na memória e torná-las acessíveis quando na realidade não são mais necessárias?

Dica de uso do especificador register

O especificador **register** de classe de armazenamento pode ser colocado antes de uma declaração de variável automática para indicar ao compilador que a variável seja conservada em um dos registradores de hardware de alta velocidade do computador. Assim, se as variáveis forem usadas frequentemente como contadores (incremento de laços), elas podem ser conservadas em registros de hardwares. Assim, o *overhead* de carregar repetidamente as variáveis na memória nos registros e armazenar os resultados novamente na memória pode ser eliminado.

Escopo de Variáveis em C

Exemplo:

```
1: #include <stdio.h>
2: #include <math.h>
3:
4: void main(void)
5: {
6:     double quantia, principal = 1000.0, taxa = .05;
7:
8:     printf("%4s%21s\n", "Ano", "Saldo na Conta");
9:
10:    for (register int ano = 1; ano <= 20; ano++){
11:        quantia = principal * pow(1+taxa, ano);
12:        printf("%4d%21.2f\n", ano, quantia);
13:    }
14:}
```

Escopo de Variáveis em C

Algumas observações sobre o código anterior:

Escopo de Variáveis em C

Algumas observações sobre o código anterior:

- ① Note que na linha 8 (primeiro **printf**) utilizamos o formato de leitura **%s**. Esse formato de leitura/escrita recebe diretamente uma *string*. No exemplo foram passados às ocorrências de **%s** as *strings* “**Ano**” e “**Saldo na Conta**”.

Escopo de Variáveis em C

Algumas observações sobre o código anterior:

- ① Note que na linha 8 (primeiro **printf**) utilizamos o formato de leitura **%s**. Esse formato de leitura/escrita recebe diretamente uma *string*. No exemplo foram passados às ocorrências de **%s** as *strings* “**Ano**” e “**Saldo na Conta**”.
- ② O compilador pode ignorar as declarações **register** em situações que não se tem um número suficiente de registros.

Escopo de Variáveis em C

Algumas observações sobre o código anterior:

- ① Note que na linha 8 (primeiro **printf**) utilizamos o formato de leitura **%s**. Esse formato de leitura/escrita recebe diretamente uma *string*. No exemplo foram passados às ocorrências de **%s** as *strings* “**Ano**” e “**Saldo na Conta**”.
- ② O compilador pode ignorar as declarações **register** em situações que não se tem um número suficiente de registros.
- ③ A palavra especificador **register** só são utilizadas em variáveis de tempo de armazenamento automático.

Escopo de Variáveis em C

Algumas observações sobre o código anterior:

- ① Note que na linha 8 (primeiro **printf**) utilizamos o formato de leitura **%s**. Esse formato de leitura/escrita recebe diretamente uma *string*. No exemplo foram passados às ocorrências de **%s** as *strings* “**Ano**” e “**Saldo na Conta**”.
- ② O compilador pode ignorar as declarações **register** em situações que não se tem um número suficiente de registros.
- ③ A palavra especificador **register** só são utilizadas em variáveis de tempo de armazenamento automático.
- ④ Frequentemente o uso do especificador **register** é desnecessário. Isto se deve ao fato de os compiladores atuais serem otimizados para entender quais variáveis são utilizadas frequentemente. Porém, é uma boa prática de programação deixar claro o seu uso.

Escopo de Variáveis em C

Fique Atento!

Fique Atento!

As palavras-chaves (especificadores) **extern** e **static** são usadas para declarar identificadores para **variáveis** e **funções** de **tempo de armazenamento estático**. Tais **variáveis/funções** existem desde o momento em que o programa começa a ser executado.

Escopo de Variáveis em C

Fique Atento!

As palavras-chaves (especificadores) **extern** e **static** são usadas para declarar identificadores para **variáveis** e **funções** de **tempo de armazenamento estático**. Tais **variáveis/funções** existem desde o momento em que o programa começa a ser executado.

Para **variáveis**, o armazenamento é alocado e inicializado quando o programa começa a ser executado. Para **funções**, o nome da função existe quando o programa começa a ser executado. Entretanto, muito embora as variáveis existam assim como os nomes das funções são conhecidos no momento de execução do programa, isso **não** significa que eles podem ser utilizados em todo o programa.

Muito Importante

As variáveis locais declaradas com a palavra-chave **static** são conhecidas apenas na função ou bloco de instrução no qual são definidas, mas diferentemente das variáveis automáticas (que são destruídas com o término do bloco ou do uso da função), as variáveis declaradas com **static conservam o seus valores** quando a função é encerrada.

Nota: Todas as variáveis numéricas de tempo de armazenamento estático são inicializadas com o valor zero (0) se não forem inicializadas explicitamente pelo programador.

Escopo de Variáveis em C

Variável Global

Chamamos de variável global, toda e qualquer variável declarada fora de qualquer função, incluindo a própria função **main** (função principal de um programa C). As variáveis globais são reconhecidas em todo o programa C. Porém, se dentro de um bloco mais interno existe a definição de uma variável com o mesmo nome (variável local), qualquer uso dentro do bloco da variável local fará uso desta variável (variável local).

Escopo de Variáveis em C

Escopo vs Tempo de Armazenamento

Os especificadores de variáveis **não alteram o escopo** das variáveis, apenas o seu tempo de armazenamento na memória. Lembre-se que o escopo de uma variável diz respeito aos lugares do código-fonte em que essas variáveis são conhecidas e não tem muita relação com o seu tempo de armazenamento na memória RAM.

Exercício: Observe o código que segue logo abaixo e descreva **exatamente** a saída esperada em tela ao executar o arquivo gerado pela compilação do código.

Escopo de Variáveis em C

```
1: #include <stdio.h>
2:
3: void a(void);
4: void b(void);
5: void c(void);
6:
7: int x = 1; /* Eu sou global */
8:
9: int main(void)
10:{ 
11:     int x = 5;
12:
13:     printf("x no bloco mais externo de main é: %d\n",
13:            x);
14:     { /* Inicio do bloco mais interno em main */
```

Escopo de Variáveis em C

```
15:     int x = 7;
16:     printf("x no bloco mais interno de main é:
16:             %d\n", x);
17: } /* Fim do bloco mais interno em main */
18:
19: printf("x no bloco mais externo em main é: %d\n",
19:         x);
20: a();
21: b();
22: c();
23: a();
24: b();
25: c();
26:
27: printf("x no bloco mais externo em main é: %d\n",
27:         x);
```

Escopo de Variáveis em C

```
28:  
29:     return 0;  
30:}  
31:  
32:void a(void)  
33:{  
34:    auto int x = 25; /* poderia ser simplesmente  
34:                      int x = 25; */  
35:    printf("\nx local %d (ao entrar em a)\n", x);  
35:    ++x;  
36:    printf("x local %d (antes de sair de a)\n", x);  
37:}
```

Escopo de Variáveis em C

```
38:void b(void)
39:{  
40:    static int x = 50;  
41:    printf("\nx local estático é %d (ao entrar em b)  
41:           \n", x);  
42:    ++x;  
43:    printf("x local estático é %d (antes de sair de b)  
43:           \n", x);  
44:}  
45:  
46:void c(void)
47:{  
48:    printf("\nx global é %d (ao entrar em c)\n", x);  
49:    x *= 10;  
50:    printf("x global é %d (antes de sair de c)\n", x);  
51:}
```

Números Aleatórios

Na estatística saber gerar números pseudo-aleatórios é uma tarefa primordial. Como a maioria dos fenômenos que estamos interessados embutem aleatoriedade, precisamos introduzir esta aleatoriedade em nossas simulações estatísticas.

```
55364966532434673230937044842076093390813045837899264488530
59540612878494730712584727673834969909231382371549952613673
1232652563373625758980260255905909525584629360808869269522
59439756367160092615839921362675272312385702543354266709983
13052003185457279658113279927621467178428282842130639223042
22428433744093574559730653033669009504982398547608318700299
85895565080425148643112316714331096600933938506831698921920
83236254471075283788235032778184679374696814061193611066637
60842490358412588467965098236949163101624497067916255266822
15841906158428512192712271932578913553199233741457449744778
14456730928886794102413530813583936647238416760159634409150
2516761334322695478902708315913920449629792770823611712459
72422373057798144966588793885733144047710652324319556265619
72432581954948839358527980414305595164540183646263687152951
67499544019170565872665828462691345626491727245520939673634
86315721485529126426598153741290671296324545804917131840909
54219271869402240557192108412443932708567865183174532372177
07254260739507714025114349109036625445628799750417185550305
80635292951025735312772410107520728001890700352492882914481
53643392347610497938392405970031031258219082832735819840846
21350867421317561336187082632809213336225267582159394058899
76672698212244322433426928409108696472661978200546337564323
84480799207160728041355839919040449910513099012769346481223
62838024388723800786521083150739204590262750405369397376009
89497877916424375732318820650415196996726355301541145798027
11084492716447904414467677658153766674254481076785043979426
00822493048963073505812790493143466927496826857168984055736
```

Números Aleatórios

Na linguagem C poderemos gerar números **pseudo-aleatórios** com o uso de função contidas no arquivo cabeçalho **stdlib.h**. Assim, o fator sorte em C poderá ser introduzido pela função **rand** fazendo:

```
i = rand();
```

Tal função gera números entre 0 e **RAND_MAX** (que é uma constante simbólica definida no arquivo cabeçalho **stdlib.h**).

O que diz o padrão ANSI C?

Segundo o padrão **ANSI C**, o valor de **RAND_MAX** deve ser menor do que **32767**, que é o valor máximo de um inteiro de dois bytes (isto é, 16 bits). Hoje em dia, isso não é mais realidade na maioria dos compiladores.

Números Aleatórios

Observação importante: Os valores gerado pela função `rand` são inteiros, isto é, do tipo `int`.

Dica

Caso se queira gerar números pseudo-aleatórios no intervalo 0 a b , podemos fazer `rand()%b`, em que b é um número inteiro. De forma geral, poderemos produzir números pseudo-aleatórios (**inteiros**) no intervalo $[a, b]$, fazendo $a + rand()%b$, em que a e b são números inteiros.

Exemplo: O exemplo que segue simula o experimento aleatório que consiste em lançar um dado equilibrado um grande número de vezes, (10000 mil vezes) observar a frequência absoluta e relativa de ocorrência de cada uma das faces. Inicialmente tente entender o código e depois implemente.

Números Aleatórios

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

#define N 10000

int main(void)
{
    int face, jogada, frequencia1 = 0, frequencia2 = 0,
        frequencia3 = 0, frequencia4 = 0, frequencia5 = 0,
        frequencia6 = 0;

    for(jogada = 1; jogada <= N; jogada++)
    {
```

Números Aleatórios

```
face = rand() % 6 + 1;

switch (face) {
    case 1:
        ++frequencia1;
        break;
    case 2:
        ++frequencia2;
        break;
    case 3:
        ++frequencia3;
        break;
    case 4:
        ++frequencia4;
        break;
}
```

Números Aleatórios

```
case 5:  
    ++frequencia5;  
    break;  
case 6:  
    ++frequencia6;  
    break;  
} /* Aqui termina o switch */  
} /* Aqui termina o laço FOR */  
printf("%s%18s%18s\n", "Face", "Frequencia",  
      "Probabilidade");  
printf("    1%18d%18.3f\n", frequencia1,  
      (float) frequencia1/N);
```

Números Aleatórios

```
    printf(" 2%18d%18.3f\n", frequencia2,
           (float) frequencia2/N);
    printf(" 3%18d%18.3f\n", frequencia3,
           (float) frequencia3/N);
    printf(" 4%18d%18.3f\n", frequencia4,
           (float) frequencia4/N);
    printf(" 5%18d%18.3f\n", frequencia5,
           (float) frequencia5/N);
    printf(" 6%18d%18.3f\n", frequencia6,
           (float) frequencia6/N);
} /* Aqui termina a função main */
```