Universidade Estadual de Maringá

Departamento de Informática - DIN

Introdução a Git/Github e Linguagem C

Autores:

Alexandre Tonet Herman
Guilherme José Monteiro Meirelles
Jean Massumi Tamura Aoyagui
Matheus Foltran Consonni
Thiago Henrique Calvi

30 de novembro de 2024

Sumário

1 Introdução

Bem-vindo(a) ao workshop "Introdução a Git/Github e Linguagem C"! Esta apostila foi desenvolvida para fornecer a você um guia prático e abrangente sobre duas ferramentas essenciais para desenvolvedores: o sistema de controle de versão Git, juntamente com a plataforma GitHub, e a linguagem de programação C.

1.1 Objetivos do Workshop

O objetivo deste workshop é oferecer uma introdução sólida e prática a esses tópicos, capacitando os participantes a:

- Comandos Básicos do CL.
- Utilização Básica do Git.
- GitHub.
- Introdução a linguagem C.

1.2 Público-Alvo

Este workshop é destinado a estudantes, desenvolvedores iniciantes, e qualquer pessoa interessada em aprender sobre controle de versão e programação em C. Não é necessário ter experiência prévia com Git, GitHub, ou C, embora conhecimentos básicos em programação possam ser úteis.

1.3 Metodologia

Nosso método de ensino combina teoria e prática. Cada seção teórica é seguida por exemplos práticos e exercícios que você pode acompanhar e executar em seu próprio computador. Encorajamos todos os participantes a praticar os comandos e códigos apresentados, garantindo assim um aprendizado mais eficaz.

Estamos entusiasmados em tê-lo conosco e esperamos que esta apostila sirva como um recurso valioso em sua jornada para se tornar proficiente em Git/Github e programação em C. Vamos começar!

2 Comando Básicos do Terminal

O **Command Line (CL)**, também conhecido como linha de comando ou terminal, é uma interface textual que permite aos usuários interagir com o sistema operacional executando comandos. É uma ferramenta poderosa para realizar uma ampla gama de tarefas, desde navegação de arquivos até a administração do sistema.

A seguir veremos alguns comando exenciais do CL tanto no Windows como no Unix/Linux.

2.1 Comandos Windows

Comando	Descrição
`dir`	Lista o conteúdo de um diretório.
`cd`	Altera o diretório de trabalho atual.
`cls`	Limpa a tela do terminal.
`copy`	Copia arquivos de um lugar para outro.
`move`	Move ou renomeia arquivos ou diretórios.
`del`	Remove arquivos.
`mkdir`	Cria um novo diretório.
`rmdir`	Remove diretórios vazios.
`type`	Mostra o conteúdo de um arquivo.
`find`	Procura por uma string de texto em um arquivo ou arquivos.
`ren`	Renomeia arquivos ou diretórios.

Tabela 1: Comandos básicos do CL no Windows

2.2 Comandos Unix/Linux

Comando	Descrição
`ls`	Lista o conteúdo de um diretório.
`cd`	Altera o diretório de trabalho atual.
`clear`	Limpa a tela do terminal.
`pwd`	Mostra o diretório de trabalho atual.
`cp`	Copia arquivos ou diretórios.
`mv`	Move ou renomeia arquivos ou diretórios.
`rm`	Remove arquivos ou diretórios.
`mkdir`	Cria um novo diretório.
`rmdir`	Remove diretórios vazios.
`touch`	Cria um novo arquivo vazio ou atualiza o timestamp de um arquivo existente.
`cat`	Mostra o conteúdo de um arquivo.
`find`	Procura por arquivos em um diretório.

Tabela 2: Comandos básicos do CL no Unix/Linux

Os comandos de linha de comando agilizam significativamente o fluxo de trabalho, uma habilidade valiosa que pode transformar o modo como você trabalha com computadores.

Vamos praticar os comandos que aprendemos até aqui com um exercício.

Exercício 1 *Utilizando o CL do seu computador crie uma pasta chamda* ***"Workshop"*** *no diretório* ***Documentos****, em seguida crie o arquivo* ***fatorial.py*** *dentro do diretório* ***Workshop****.*

1. *Abra o terminal do seu computador.*
2. *Utilize o comando* `dir` *no Windows e* `ls` *para Unix/Linux para listar os aquivos e pastas no diretório atual.*
3. *Com o camando* `cd Documentos` *entre na pasta Documentos.*
4. *Utilize o comando* `mkdir Workshop` *para criar a pasta Workshop dentro de Documentos.*
5. *Entre na pasta que acabou de criar.*
6. *Por fim crie o arquivo* ***fatorial.py*** *utilizando* `type nul > fatorial.py` *no Windows e* `touch fatorial.py` *no Linux.*

3 Conceitos Básicos do Git

Git é um sistema de controle de versão distribuído amplamente utilizado por desenvolvedores para gerenciar e acompanhar alterações no código-fonte ao longo do tempo. Ele permite que múltiplos desenvolvedores trabalhem no mesmo projeto de maneira eficiente e organizada, mantendo um histórico completo de todas as alterações feitas.

Git foi criado por Linus Torvalds, o mesmo desenvolvedor responsável pela criação do kernel do Linux. Torvalds iniciou o desenvolvimento do Git em abril de 2005, como uma ferramenta para auxiliar no gerenciamento do desenvolvimento do kernel do Linux.

Antes do Git, o projeto do kernel do Linux utilizava um sistema de controle de versão proprietário chamado BitKeeper. No entanto, devido a questões de licenciamento e restrições, a comunidade Linux precisou de uma alternativa livre e aberta. A exigência era clara: o novo sistema deveria ser rápido, simples, escalável e suportar desenvolvimento distribuído. Em resposta a essas necessidades, Torvalds desenvolveu o Git, com a ajuda de outros colaboradores, para atender aos requisitos específicos do desenvolvimento do kernel e superar as limitações das ferramentas existentes.

3.1 O que é Controle de Versão?

Controle de versão é uma metodologia que permite acompanhar e gerenciar alterações em arquivos ao longo do tempo. Com o controle de versão, você pode:

- Manter um registro detalhado de todas as modificações feitas nos arquivos do projeto, incluindo quem fez cada alteração e quando.
- Visualizar as diferenças entre diferentes versões de um arquivo ou do código-fonte completo, facilitando a compreensão das mudanças realizadas ao longo do tempo.
- Retornar a qualquer momento para uma versão anterior do projeto, útil para desfazer mudanças indesejadas ou corrigir problemas que surgiram.
- Permitir que múltiplos desenvolvedores trabalhem no mesmo projeto sem conflitos, através da utilização de branches (ramificações) que podem ser integradas posteriormente.
- Gerar automaticamente um histórico de todas as atividades realizadas no projeto, incluindo a adição de novos recursos, correções de bugs, e implementação de melhorias.
- Promover uma colaboração mais eficiente entre membros da equipe, possibilitando revisões de código, feedbacks e validações antes da integração de novas funcionalidades ao projeto principal.
- Proteger os arquivos contra perdas acidentais, pois todas as versões anteriores permanecem registradas e acessíveis.
- Criar branches separadas para desenvolver novas funcionalidades sem impactar a versão principal do projeto, permitindo testes e validações antes da integração final.
- Integrar o controle de versão com outras ferramentas e processos de desenvolvimento, como integração contínua (CI), para automatizar testes e garantir a qualidade do código.

- Preparar e distribuir releases estáveis do software, utilizando tags para marcar versões específicas do código que foram aprovadas para distribuição.

3.2 Por que usar o Git?

Git é uma das ferramentas de controle de versão mais populares devido às suas características:

- Usa SHA-1 para nomear e identificar objetos (arquivos, diretórios, commits), garantindo a integridade dos dados.
- Proporciona operações rápidas e eficientes, mesmo em projetos grandes.
- Cada desenvolvedor possui uma cópia completa do repositório, permitindo trabalho offline e sincronização quando necessário.
- Suporta diversos fluxos de trabalho e estilos de desenvolvimento.

3.3 Instalação do Git

Para começar a usar o Git, você precisa instalá-lo em seu computador. As instruções de instalação podem varia dependento do seu sistema operacional.

Para **Windows**, você pode baixar o instalador do Git a partir do site oficial (https://www.git-scm.com). Ou caso prefira é possivel instalá-lo pelo PowerShell utilizando o seguinte comando: `winget install -id Git.Git -e -source winget`

No **MacOs**, o Git pode ser instalado utilizando um do seguites gerenciadores de pacotes:

- Homebrew (`brew install git`)
- MacPorts (`sudo port install git`)

Também está disponivel um instalador para **MacOs**.

A instalação no **Linux** pode ser feita utilizando o gerenciador de pacotes da sua distribuição (por exemplo, sudo `apt-get install git` no Ubuntu). O endereço (https://www.git-scm.com/download/linux) contem os comandos de instalação para outras distribuições Linux.

3.4 Configuração Inicial

Após instalar o Git, configure seu nome de usúario e endereço de e-mail, esses dados serão usados em seus commits:

```
git config --global user.name "Seu nome"
git config --global user.email "seuemail@exemplo.com"
```

3.5 Comandos Básicos

A seguir, apresentamos alguns comandos básicos do Git que você usará frequentemente:

- **git init**: Inicializa um novo repositório Git.
- **git clone *url***: Clona um repositório existente.
- **git status**: Mostra o status das alterações no repositório.
- **git add *arquivo***: Adiciona um arquivo ao índice (staging area).
- **git commit -m "mensagem"**: Confirma as alterações adicionadas com uma mensagem descritiva.
- **git push**: Envia os commits locais para um repositório remoto.
- **git pull**: Atualiza o repositório local com as alterações do repositório remoto.
- **git branch**: Lista, cria ou deleta branches.
- **git checkout *branch***: Alterna entre branches.
- **git merge *branch***: Mescla uma branch com a branch atual.

3.6 Fluxo de Trabalho Básico

O fluxo de trabalho básico com Git gerelmente segue estes seguintes passos:

1. Inicialize o repositório local usando `git init`.
2. Faça alterações em seu código.
3. Use `git add` *`arquivo`* para adicionar as alterações ao índice.
4. Use `git commit -m` *`mensagem`* para salvar suas alterações no histórico do repositório.
5. Use `git push` para enviar seus commits para o repositório remoto (por exemplo, no GitHub)

4 Conceitos Básicos do GitHub

O GitHub é uma platadoma baseada em cloud onde você pode armazenar, compartilhar e trabalhar com outros colaboradores escrevendo código.

Foi fundado em 2008 por Tom Preston-Werner, Chris Wanstrath, PJ Hyett e Scott Chacon. A ideia surgiu da necessidade de um sistema de controle de versão descentralizado e fácil de usar, baseado em Git, que havia sido desenvolvido por Linus Torvalds em 2005.

O GitHub rapidamente ganhou popularidade entre desenvolvedores de software open-source devido à sua interface amigável e às funcionalidades colaborativas que oferecia.

Em 2018 o GitHub foi comprado pela Microsoft por 7,5 bilhões de dólares. Sob a propriedade da Microsoft, GitHub introduziu várias novas funcionalidades e melhorias, incluindo o GitHub Actions para CI/CD e o GitHub Copilot, uma ferramenta de inteligência artificial para assistência na programação.

4.1 Criando uma conta no GitHub

Para criar uma conta no GitHub, siga os passos abaixo:

1. Vá para o site oficial do GitHub: https://github.com/.
2. No canto superior direito da página inicial, você verá duas opções: **Sign in (entrar)** e **Sign up (inscrever-se)**.
3. Se você já possui um conta no GitHub, clique em **Sign in**, insira seu email e sua senha para entrar na sua conta. Caso não possua, siga para a próxima etapa.
4. Clique em **Sign up**:
   - 4.1 Digite um e-mail acessível. O e-mail deve ser acessível pois você precisará confirmar a criação da conta através de um código enviado para ele.
   - 4.2 Digite uma senha segura para sua conta.
   - 4.3 Informe um nome de usuário de sua preferência. Este será o nome pelo qual outras pessoas encontrarão você no GitHub. É recomendado escolher um nome adequado para fins acadêmicos e profissionais.

     Obs.:Você não poderá mudar seu nome de usuário depois de concluir a criação da conta.
   - 4.4 Faça a verificação para confirmar que você não é um robo.
   - 4.5 Por último, digite o código que foi enviado para o seu e-mail para concluir o criação da sua conta.

4.2 Beneficios para Estudantes

O GitHub oferece gratuitamente uma variedade de ferramentas e serviços enquanto você for estudante vinculado a uma instituição de ensino. Este programa, conhecido como GitHub Education, visa apoiar o aprendizado e o desenvolvimento de habilidades técnicas entre estudantes.

O GitHub Student Developer Pack é um dos principais benefícios deste programa. Ele inclui acesso gratuito a diversas ferramentas e serviços que normalmente seriam pagos,

proporcionando aos estudantes uma oportunidade única de explorar e utilizar recursos profissionais de desenvolvimento de software. Entre os benefícios oferecidos, destacam-se:

- Repositórios privados ilimitados: Permite aos estudantes desenvolver projetos de forma privada, ideal para trabalhos acadêmicos e portfólios pessoais.
- GitHub Copilot: Uma ferramenta de inteligência artificial que auxilia na escrita de código, oferecendo sugestões em tempo real baseadas no contexto do projeto.
- GitHub Pro: Versão aprimorada do GitHub com recursos adicionais, como análise de código avançada e páginas do GitHub.
- Acesso a plataformas de aprendizado: Inclui assinaturas gratuitas ou com desconto para plataformas de cursos online como Educative, DataCamp e FrontendMasters.
- Créditos em serviços de nuvem: Oferece créditos para uso em plataformas como DigitalOcean, Microsoft Azure e Heroku, permitindo aos estudantes hospedar e implantar suas aplicações.
- Ferramentas de desenvolvimento: Acesso a IDEs profissionais como JetBrains.

Vejá todos os benefícios oferecidos em: https://education.github.com/pack/offers

Além desses benefícios diretos, o GitHub Education também promove iniciativas como o GitHub Campus Program, que oferece suporte adicional para instituições de ensino, e o GitHub Classroom, uma ferramenta que facilita a distribuição e avaliação de tarefas de programação em ambientes educacionais.

Esses recursos não apenas fornecem aos estudantes acesso a ferramentas profissionais, mas também os expõem a práticas da indústria, preparando-os melhor para futuras carreiras em tecnologia. O programa incentiva a colaboração, o desenvolvimento de portfólio e a participação em projetos de código aberto, aspectos cada vez mais valorizados no mercado de trabalho de tecnologia.

4.3 Como se inscrever no GitHub Student Developer Pack

Para se beneficiar desses recursos, siga os passos abaixo para se inscrever no GitHub Student Developer Pack:

1. Crie uma conta no GitHub, caso ainda não tenha uma.
2. Acesse a página do GitHub Education (https://education.github.com/).
3. Clique em "Join GitHub Education".
4. Faça login com sua conta do GitHub.
5. Preencha o formulário de inscrição com suas informações acadêmicas, incluindo:
   - Nome da sua instituição de ensino
   - Seu status acadêmico (por exemplo, estudante de graduação)
   - Ano previsto de formatura

- Endereço de e-mail acadêmico (se disponível)

6. Faça o upload de um documento que comprove seu vínculo acadêmico. Isso pode ser:
   - Uma foto da sua carteirinha de estudante
   - Um histórico escolar recente
   - Uma carta de aceitação da instituição
   - Comprovante de matrícula
7. Explique brevemente como você planeja usar o GitHub.
8. Revise e aceite os termos de uso do GitHub Education.
9. Envie sua solicitação.

4.4 Conexão SSH com o GitHub

O uso de SSH (Secure Shell) para interagir com o GitHub é uma prática altamente recomendada e importante para desenvolvedores. Este método oferece várias vantagens significativas em termos de segurança, eficiência e conveniência.

4.4.1 Segurança Aprimorada

- **Criptografia Robusta:** SSH utiliza criptografia forte para proteger todas as comunicações entre seu computador e o GitHub, garantindo que seus dados e código permaneçam seguros durante a transmissão.
- **Autenticação Segura:** Ao contrário da autenticação baseada em senha, o SSH usa um par de chaves (pública e privada), o que é significativamente mais seguro e resistente a ataques.
- **Proteção contra Ataques de Intermediários:** SSH ajuda a prevenir ataques do tipo "man-in-the-middle", onde um atacante pode interceptar comunicações entre você e o GitHub.

4.4.2 Conveniência e Eficiência

- **Autenticação sem Senha:** Uma vez configurado, o SSH permite que você se conecte ao GitHub sem inserir sua senha repetidamente, tornando o processo mais rápido e conveniente.
- **Gestão de Múltiplas Contas:** SSH facilita o gerenciamento de múltiplas contas GitHub em uma única máquina, permitindo o uso de diferentes chaves para diferentes contas.
- **Automação Simplificada:** Para tarefas automatizadas e scripts, o SSH é mais fácil de implementar e gerenciar do que autenticação baseada em senha.

4.4.3 Conformidade e Boas Práticas

- **Padrão da Indústria:** O uso de SSH é uma prática padrão na indústria de desenvolvimento de software, especialmente para operações em repositórios remotos.
- **Conformidade com Políticas de Segurança:** Muitas organizações exigem o uso de SSH para acessar sistemas remotos, incluindo repositórios de código.
- **Auditoria e Rastreabilidade:** SSH facilita o rastreamento de quem fez o quê, pois cada chave pode ser associada a um usuário específico.

4.4.4 Desempenho

- **Conexões Mais Rápidas:** SSH geralmente oferece melhor desempenho em comparação com conexões HTTPS, especialmente para operações frequentes como push e pull.
- **Menor Sobrecarga de Rede:** A natureza da conexão SSH pode resultar em menor tráfego de rede para certas operações.

4.4.5 Conclusão

O uso de SSH no GitHub não é apenas uma questão de preferência, mas uma prática essencial para desenvolvimento seguro e eficiente. Ele oferece um equilíbrio ideal entre segurança, conveniência e desempenho, tornando-se uma ferramenta indispensável para desenvolvedores modernos. Adotar o SSH no seu fluxo de trabalho com o GitHub é um passo importante para melhorar a segurança de seus projetos e otimizar seu processo de desenvolvimento.

4.4.6 Configurando a conexão SSH com o GitHub

1. Verifique se você já tem uma chave SSH:

```
ls -al ~/.ssh
```

   Se você ver arquivos como `id_rsa.pub` ou `id_ed25519.pub`, você já tem uma chave SSH.

2. Se não tiver uma chave SSH, gere uma nova:

```
ssh-keygen -t ed25519 -C "seu_email@exemplo.com"
```

   Pressione Enter para aceitar o local padrão do arquivo. Digite uma senha segura quando solicitado.

3. Inicie o ssh-agent em segundo plano:

   (a) Linux/macOS:

```
        eval "$(ssh-agent -s)"
```

   (b) Windows (Git Bash):

```
        eval `ssh-agent -s`
```

4. Adicione sua chave SSH ao ssh-agent:

```
ssh-add ~/.ssh/id_ed25519
```

5. Copie a chave SSH para sua área de transferência:

   - Linux:

     ```
     xclip -selection clipboard < ~/.ssh/id_ed25519.pub
     ```

     Se `xclip` não estiver instalado, use:

     ```
     cat ~/.ssh/id_ed25519.pub
     ```

     E copie manualmente a saída.

   - MacOS:

     ```
     pbcopy < ~/.ssh/id_ed25519.pub
     ```

   - Windows (PowerShell):

     ```
     Get-Content ~/.ssh/id_ed25519.pub | Set-Clipboard
     ```

     Ou no Command Prompt:

     ```
     type %userprofile%\.ssh\id_ed25519.pub | clip
     ```

6. Adicione a chave SSH à sua conta do GitHub:

   - Vá para GitHub.com e faça login.
   - Clique na sua foto de perfil e selecione "Settings".
   - No menu lateral esquerdo, clique em "SSH and GPG keys".
   - Clique em "New SSH key"ou "Add SSH key".
   - Cole sua chave no campo "Key".
   - Dê um título descritivo à sua chave no campo "Title".
   - Clique em "Add SSH key".

7. Teste sua conexão:

```
ssh -T git@github.com
```

   Você pode ver um aviso sobre a autenticidade do host. Digite 'yes' para continuar.

8. Se tudo estiver configurado corretamente, você verá uma mensagem de boas-vindas do GitHub.

Nota: Para Windows, é recomendável usar Git Bash para executar comandos Unix-like.

Agora você pode usar URLs SSH para suas operações do Git com o GitHub. Lembre-se de usar a URL SSH ao clonar novos repositórios ou alterar a URL remota de repositórios existentes.

4.5 Personal Acesses Token (classic)

O Personal Access Token (PAT) clássico do GitHub é uma forma de autenticação que permite que usuários acessem a API do GitHub e realizem operações em suas contas sem precisar usar a senha diretamente.

O PAT serve como um substituto para a senha ao fazer operações que exigem autenticação, como acessar repositórios, clonar projetos ou usar a API. Ao criar um PAT, você pode definir quais permissões ou "escopos"o token terá, como acesso a repositórios privados, gerenciamento de gists, etc.

4.5.1 Criando um PAT

1. Acesse as configurações da sua conta do GitHub.
2. Acesse as configurações de desenvolvedor.
3. Em seguida, acesse **Personal Access Token** no menu lateral e clique em **Tokens (classic)**.
4. Clique em **Generate new token > Generate new token (classic)** no canto superior direito.
5. Em "Note", escreva o propósito do token.
6. Defina o tempo de vida do token em **Expiration**.
7. Em **Select scopes**, marque a opção **repo**.
8. Por fim, clique em **Create token** e copie o token gerado.

Agora vamos configurar o Git para usar o PAT.

1. Abra o terminal.
2. Use o seguinte comando para armazenar as credenciais:

```
1         git config --global credential.helper cache
2
```

Isso armazenará suas credenciais na memória por um tempo.

3. Na próxima vez que você realizar uma operação que exija autenticação (como ‘git push‘ ou ‘git clone‘), insira seu nome de usuário do GitHub e, em vez de sua senha, insira o PAT.

Outra forma é configurar diretamente o repositório. Se você deseja que o Git use o PAT apenas para um repositório específico, pode configurar as credenciais diretamente na URL do repositório:

```
1     git clone https://<username>:<token>@github.com/<username>/<
    repository>.git
```

Substitua ‘<username>‘, ‘<token>‘ (seu PAT) e ‘<repository>‘ pelos valores apropriados.

5 História da Linguagem C

A linguagem de programação C foi desenvolvida no início dos anos 1970 no Bell Labs, uma divisão da AT&T. Seu criador principal, Dennis Ritchie, trabalhou em conjunto com Brian Kernighan e outros colegas para desenvolver essa linguagem, que se tornaria uma das mais influentes na história da computação. Origem e Desenvolvimento

A necessidade de uma nova linguagem surgiu da criação do sistema operacional UNIX, que inicialmente foi escrito em Assembly. O Assembly, embora eficiente, era complexo e difícil de portar entre diferentes tipos de hardware. Para resolver esses problemas, Ken Thompson e Dennis Ritchie começaram a trabalhar em uma linguagem intermediária chamada B, que foi uma derivação da linguagem BCPL (Basic Combined Programming Language), criada por Martin Richards.

No entanto, a linguagem B tinha suas limitações, especialmente em termos de capacidade de dados. Em resposta a isso, Dennis Ritchie desenvolveu a linguagem C em 1972. A nova linguagem oferecia melhor manipulação de tipos de dados e estruturas, tornando-se mais adequada para o desenvolvimento de sistemas operacionais.

5.1 Adoção e Popularização

A primeira implementação significativa de C foi a reescrita do UNIX em 1973. Essa reescrita demonstrou a portabilidade e a eficiência da linguagem, uma vez que o UNIX podia agora ser facilmente adaptado a diferentes máquinas. A partir daí, o uso de C se expandiu rapidamente, não apenas em sistemas operacionais, mas também em outras áreas da computação.

Em 1978, Brian Kernighan e Dennis Ritchie publicaram o livro "The C Programming Language", frequentemente chamado de "K&R". Esse livro se tornou a principal referência para a linguagem e ajudou a padronizá-la. A linguagem descrita no livro era uma versão de C que se tornaria conhecida como "K&R C".

5.2 Padronização

Com o crescimento da popularidade de C, surgiu a necessidade de uma padronização formal. Em 1983, o American National Standards Institute (ANSI) formou um comitê, conhecido como X3J11, para criar uma especificação padrão para a linguagem. O resultado foi a publicação do ANSI C em 1989. Esse padrão foi posteriormente adotado pela International Organization for Standardization (ISO), sendo publicado como ISO C em 1990.

5.3 Evolução

Desde sua padronização inicial, a linguagem C passou por diversas revisões. Notáveis entre elas são o C99, que introduziu várias melhorias como novas funções de biblioteca, suporte a comentários de linha única e maior flexibilidade na declaração de variáveis, e o C11, que trouxe melhorias de desempenho e suporte a multithreading.

5.4 Impacto e Legado

A linguagem C teve um impacto profundo na computação. Muitas linguagens modernas, como C++, C#, Java e JavaScript, foram influenciadas por sua sintaxe e conceitos.

Além disso, C continua sendo amplamente utilizada em sistemas embarcados, sistemas operacionais e outras áreas onde a eficiência e o controle de baixo nível do hardware são críticos.

A simplicidade, a eficiência e a flexibilidade de C garantiram seu lugar como uma das linguagens de programação mais importantes e duradouras. Seu legado perdura na educação em ciência da computação, na engenharia de software e no desenvolvimento de tecnologia.

6 Visão Geral

6.1 Utilidades

A linguagem de programação C está presente em diversos contextos. Desde a robótica e sistemas embarcados até ser a base de sistemas operacionais e outras linguagens, esta tecnologia é amplamente utilizada.

6.2 Contraste entre C e outras linguagens

Quando comparado com outras linguagens, o C pode possuir algumas diferenças.

1. Compilação: ao contrário de outras linguagens, o C é compilado. Isto significa que, ao concluir uma versão do código deve-se efetuar o processo de compilação (utilizando-se de algum compilador, como o gcc). À conclusão deste processo, estará gerado um arquivo executável do programa. Isto significa que o algoritmo implementado será encontrado no formato em que o computador executará o código próximo ao nível de máquina, o que também significa que este formato é rígido; o computador sempre executará o código de uma mesma forma, sem possíveis alterações em tempo de execução, ao contrário do que ocorre em linguagens interpretadas.

2. Diferenças de sintaxe: o que é necessário ou não para efetuar uma estrutura ou ação pode ser diferente em C em relação a outras linguagens. Exemplo: é necessário utilizar o ";"ao final de diversas instruções

3. Diferenças de tipagem: ao contrário de linguagens fracamente tipadas (geralmente linguagens interpretadas), o C é fortemente tipado. Isso significa que ele não tolera mudança da natureza de um dado, isto é, caso um identificador de dado (variável) seja declarado de um tipo (por exemplo inteiro), o C não tolerará a atribuição de um valor de outro tipo (por exemplo float) a este identificador, a não ser que haja uma conversão do tipo de valor explicitamente antes de ser realizada a atribuição.

7 Estrutura de um programa em C

7.1 Blocos de código

Na linguagem de programação C, o código é estruturado em blocos de código. Estes blocos de código são delimitados por "{"antes do início do bloco, e por "}"após o final do bloco. Além disso, ao final de cada instrução deverá haver um ";".

Exemplo de bloco de código:

```
1 {
2     int a;
3     a = 3;
4     int b = 4;
5     a = a + b;
6 }
```

A identação não é obrigatória, podem haver mais de uma instrução em uma linha ou uma instrução em mais de uma linha. Assim, o bloco de código acima pode ser reescrito da seguinte forma, de modo a permanecer executando as mesmas instruções:

```
1 {int a; a = 3; int b = 4; a =
2 a + b;}
```

Uma declaração de variável, como em "int a;"é uma instrução em que se é criada a variável em questão. Caso uma variável seja criada dentro de um bloco de código delimitado por "{"e "}", esta variável estará no escopo local do bloco em que está, ou seja, será eliminada ao final da execução do bloco. Segue um exemplo em que um erro envolvendo esta questão ocorre:

```
1 {
2     int a;
3     a = 3;
4     int b;
5     b = 4;
6     a = a + b;
7 }
8 a = a + 1;
```

Neste exemplo, ocorre um erro no código, pois a variável "a"existe apenas localmente dentro do bloco de código, pois ela foi declarada dentro deste bloco. Assim, quando há uma instrução que tenta acessar a variável "a"fora do bloco, na linha 8 (em a = a + 1), esta ação não pode ocorrer, pois esta variável não existe neste escopo.

O erro não ocorreria se o trecho de código fosse reescrito da seguinte forma:

```
1 int a;
2 {
3     a = 3;
4     int b;
5     b = 4;
6     a = a + b;
7 }
8 a = a + 1;
```

Desta forma o erro não ocorre pois a declaração da variável (linha 1) se encontra fora do bloco de código, assim como a instrução que tenta acessá-la (linha 8). Assim sendo, a variável existe dentro do escopo em que tenta-se acessá-la e portanto esta instrução é bem-sucedida.

7.2 Atribuição de valor a uma variável

A atribuição de valor a uma variável é a instrução que armazena um valor na variável em questão. Esta operação é da forma "variavel = valor", em que valor pode ser outra variável, uma constante (por exemplo 5.0), o valor de retorno de uma função ou o resultado de operações entre variáveis e/ou constantes e/ou valores de retorno de funções.

A atribuição de uma variável pode ocorrer juntamente à sua declaração, porém, considerando uma mesma variável, a atribuição pode ocorrer diversas vezes ao longo do código, enquanto a declaração ocorre apenas uma vez, antes (ou juntamente) da primeira atribuição.

Segue um exemplo em que a declaração e atribuição de uma variável ocorrem na mesma instrução:

```
1 int a = 3.0;
```

7.3 Includes

Ao escrever um código na linguagem C, é possível incluir nele outras partes de código (por exemplo funções, que veremos mais adiante) de bibliotecas ou outros códigos em C (que possivelmente o mesmo autor que inclui escreveu).

Por exemplo, na blioteca padrão "standard input-output" (entrada e saída padrão, ou em sigla, stdio), podemos encontrar a função de entrada scanf() e a função de saída printf(). Por meio da linha de código a seguir podemos incluí-las no código, isto é, possibilitar o uso das suas funções no código.

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

Ao escrever esta linha de código no início do programa, pode-se utilizar os recursos da biblioteca stdio na adiante. Assim sendo, para incluir os recursos de uma biblioteca utiliza-se

```
1 #include <biblioteca.h>
```

em que biblioteca é a sequência de caracteres utilizada para se referir à biblioteca desejada. Caso deseja-se incluir os recursos de outro arquivo em código-fonte C, utiliza-se:

```
1 #include "nomearquivo.c"
```

em que nomearquivo é o nome do arquivo código-fonte do qual se tem como objetivo incluir os recursos.

7.4 Tipos de dados primitivos

Um **tipo de dado primitivo** é um formato em que se armazena um valor. Em C, existe o tipo primitivo **char**, que armazena um caracter (porém também pode ser usado para armazenar um número inteiro de -128 a 127), e os tipos **short**, **int**, **long**, **float** e **double**, que armazenam números.

Os tipos **short**, **int** e **long** armazenam números inteiros (por exemplo, "5", "37", "40") e seus valores mínimos e máximos são, respectivamente:
short: -32768 e 32767
int: -2147483648 e 2147483647
long: -9223372036854775808 e 9223372036854775807

Os tipos **float** e **double** armazenam números com **parte do valor fracionária** (por exemplo "3,5", "4,0" e "8,125"). Em C, a vírgula que separa a parte inteira da parte

fracionária não é usada. Em seu lugar, utiliza-se o ponto. Além disso para distinguir uma constante float de uma constante double, utiliza-se "f"' ao final de uma constante float.

Exemplos: “3,5” se escreve “3.5f” para float e “3.5” para double, “8,125” se escreve “8.125f” para float e “8.125” para double, e “4,0” se escreve “4.0f” para float e “4.0” para double.

O tipo double pode armazenar valores maiores e com maior precisão quando comparado ao tipo float.

O caracter "a", por exemplo, deve ser escrito entre apóstrofos em C, da seguinte forma: ’a’. Caso deseja-se armazenar este caracter em uma variável, esta variável será do tipo char. Uma sequência de caracteres é chamada de string, como por exemplo “Hello World!”. Essa string deve ser escrita entre aspas, como segue: "Hello World!". Armazena-se uma string em um vetor ou array de char (char[ ]).

7.5 Operações

Operações são ações que tomam como entrada dois ou mais valores e produzem um valor de saída. Em C existem as funções aritméticas, lógicas e relacionais.

7.5.1 Operações aritméticas

As operações aritméticas em C são a adição (+), subtração (-), multiplicação (*), divisão (/) e módulo ou resto da divisão (%), esta última tendo de ser realizada entre dois inteiros, necessariamente.

No caso da divisão, caso ambos os operandos sejam inteiros, será obtido o resultado da divisão inteira entre eles. Por exemplo $5/2 = 2$ (em vez de 2.5) porém, caso pelo menos um dos operandos seja float/double, o resultado será float/double. Por exemplo $5/2.0 =$ 2.5f ou 2.5.

As operações + e - são realizados após a realização das operações *, / e %.

7.5.2 Operações relacionais

Em C, 0 é interpretado como valor lógico falso e números diferentes de 0 são interpretados como valor lógico verdadeiro. Operações relacionais geram como resultado valor lógico verdadeiro ou valor lógico falso.

Os operadores lógicos são x igual a y (x == y), x diferente de y (x != y), x menor que y (x < y), x menor ou igual a y (x <= y), x maior que y (x > y), e x maior ou igual a y (x >= y).

Por exemplo 5 >= 2 produz 1 (valor lógico verdadeiro), pois 5 é maior ou igual a 2, e 1 >= 12 produz 0 (valor lógico falso), pois 1 não é maior ou igual a 12.

7.6 Funções

Uma função em C é declarada por meio de uma expressão conforme o modelo abaixo.

```
1 <tipo_retorno> <nome> (<tipo_parametro> <nome_parametro>){
2     bloco_de_codigo
3 }
```

A função poderá não retornar um valor; neste caso o campo do tipo de retorno é "void". Caso o tipo de retorno da função não seja void, deverá haver um "return"acessível no bloco de código.

A função printf(), por exemplo, exibe no terminal o conteúdo em string contido no argumento passado (parte entre parêntesis).

Exemplos de funções em C pode ser encontrados nos trechos de código abaixo:

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

No exemplo a seguir, printf("%d", soma(a, b)) exibe o retorno de soma(a, b). O "%d"dentro das aspas antes da vírgula indica que o valor será exibido no formato de inteiro, e o "soma(a, b)"depois da vírgula indica que o valor exibido será o valor do retorno da função soma passando-se como argumentos os inteiros "a"e "b".

```
void printf_soma(int a, int b){
    printf("%d", soma(a, b));
}
```

```
void hello_world(){
    printf("Hello, World!");
}
```

7.7 A função main

A função main é o corpo principal do código em C. Assim, a função main é obrigatória, o programa começa no início da função main e termina ao final da função main. Logo, tudo o que ocorre no programa ou acontece na main, ou é chamado direta ou indiretamente pela função main.

As funções utilizadas no programa devem estar declaradas antes da função main ou incluídas no código através de um include. Desta forma, exemplificando a função main, juntando os conceitos anteriores e demonstrando como pode ser um programa completo em C, segue o código:

```
#include <stdio.h>

int soma(int a, int b){
    return (a + b);
}
void printf_soma(int a, int b){
    printf("%d", soma(a, b));
}
void hello_world(){
    printf("Hello, World!");
}
void main(){
    hello_world();
    printf("\n");
    int x, y;
    x = 2;
    y = 3;
    printf_soma(x, y);
    printf("\n");
    int z = soma(soma(x, x), soma(x, y));
    z = z + 1;
    printf("%d\n", z);
}
```

Ao final da execução deste código, será exibido a seguinte saída:

```
1 Hello, World!
2 5
3 10
```

Na linha 1 da saída, “Hello, World!” é a saída da função printf(), passando-se como argumento a string "Hello, World!". Esta execução da função printf() ocorre dentro da função hello_world(), chamada na linha 11 do código. Na linha 2, 5 é a saída da função printf() dentro de printf_soma(). Por, fim, na linha 3, 10 é a saída da função printf() passando-se como argumento a variável z, na linha 19, na função main().

Quando chamada a função printf() passando-se como argumento "\n", printf() adiciona uma quebra de linha à saída.

7.8 Entrada e saída

Na linguagem C, para que haja entrada e saída de dados via linha de comando, é possível utilizar o cabeçalho da biblioteca padrão do C stdio.h (standard input-output header). Desta forma, pode-se utilizar a função printf() para saída de dados (exibição de dados no formato textual) e a função scanf() para entrada de dados (obtenção de dados digitados pelo usuário) caso o cabeçalho “<stdio.h>” tenha sido incluído no início do código. Ou seja, para utilizar as funções printf() e scanf(), é necessário haver a seguinte linha de código ao início do código-fonte em C:

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

Em ambas as funções, os argumentos são uma string formatada, seguida das variáveis ou endereços das variáveis na ordem em que elas aparecem na entrada ou saída. A formatação da string se dá pela combinação dos caracteres “\” ou “%” com outros caracteres. Por exemplo, “\n” significa quebra de linha e “%d” significa a referência a um dado do tipo número inteiro na entrada ou saída. Desta forma, considerando que a variável x seja um dado do tipo número inteiro, para exibí-lo na saída do programa, pode-se utilizar a linha de código a seguir:

```
1 printf("%d", x);
```

De forma similar, para que a variável x receba um valor digitado pelo usuário, é possível utilizar a linha de código a seguir:

```
1 scanf("%d", &x);
```

Neste caso, a variável x é precedida pelo símbolo & pois deseja-se destinar o valor inserido pelo usuário ao endereço da variável x. Sempre que desejar-se referenciar o endereço de uma variável, utiliza-se & antes da variável em questão.

Caso seja desejado exibir diversos dados com apenas uma chamada da função printf(), é possível fazê-lo utilizando várias referências aos dados (por exemplo “%d”) na string formatada na forma desejada, além de passar como argumento todos os valores que deseja-se exibir. Por exemplo, para exibir “<valor de x> + <valor de y> + <valor de z> = <resultado da soma de x, y e z>, utiliza-se a seguinte linha de código:

```
1 printf("%d + %d + %d = %d", x, y, z, (x + y + z));
```

Note que x, y, z e (x + y + z) aparecerão nesta ordem, pois é a ordem com que foram passados nos argumentos.

Como visto, %d é utilizado para indicar um dado do tipo inteiro na string formatada passado como argumento no printf() ou scanf(). Além desta formatação, também é possível utilizar %f para float ou double, %s para string e %c para caracter.

8 Estruturas

8.1 Estruturas Condicionais

8.1.1 If

A estrutura if avalia uma condição booleana (verdadeira ou falsa). Se a condição for verdadeira, o bloco de código dentro do if é executado. Caso contrário, o bloco é ignorado.

Sintaxe:

```
if(condicao){
    // Codigo a ser executado
}
```

Exemplo:

```
#include <stdio.h>

int idade = 20;
if (idade >= 18) {
    printf("Voce e maior de idade.");
}
```

8.1.2 Else

A estrutura else é usada para fornecer uma alternativa caso a condição no if seja falsa. Ela é opcional e deve ser colocada após o bloco if.

Sintaxe:

```
if(condicao){
    // Codigo executado se a condicao for verdadeira
} else {
    // Codigo executado se a condicao for falsa
}
```

Exemplo:

```
#include <stdio.h>

int idade = 16;
if (idade >= 18) {
    printf("Voce e maior de idade.\n");
} else {
    printf("Voce e menor de idade.\n");
}
```

8.1.3 Else if

O else if é usado quando você precisa verificar múltiplas condições. Ele permite adicionar condições adicionais a serem verificadas se a condição anterior for falsa.

Sintaxe:

```c
if (condicao1) {
    // Codigo executado se a condicao1 for verdadeira
} else if (condicao2) {
    // Codigo executado se a condicao1 for falsa e a condicao2 for verdadeira
} else {
    // Codigo executado se todas as condicoes anteriores forem falsas
}
```

Exemplo:

```c
#include <stdio.h>

int nota = 85;
if (nota >= 90) {
    printf("Nota A.\n");
} else if (nota >= 80) {
    printf("Nota B.\n");
} else if (nota >= 70) {
    printf("Nota C.\n");
} else {
    printf("Nota D.\n");
}
```

8.1.4 Switch

O switch em C é uma estrutura condicional que permite escolher entre várias opções de execução, dependendo do valor de uma variável ou expressão. Ele é útil quando você tem uma variável que pode assumir vários valores e deseja executar diferentes blocos de código com base em cada valor.

Sintaxe:

```c
switch (expressao) {
    case valor1:
        // Codigo a ser executado se expressao == valor1
        break;
    case valor2:
        // Codigo a ser executado se expressao == valor2
        break;
    // Outros casos...
    default:
        // Codigo a ser executado se expressao nao corresponder a nenhum caso
        break;
}
```

Detalhes:

- expressao: Uma expressão que é avaliada e cujo resultado é comparado com os valores especificados em cada case.
- case: Cada case é seguido por um valor constante (geralmente um int ou char) e dois pontos. Se o valor da expressao corresponder a este valor, o código associado ao case é executado.
- break: O comando break é usado para sair do switch após a execução de um case. Sem o break, a execução continuará para os próximos casos, um comportamento chamado de "fall-through".

- default: O bloco default é opcional e é executado se nenhum dos case for correspondente ao valor da expressao.

Exemplo:

```c
int dia = 3;

switch (dia) {
    case 1:
        printf("Domingo\n");
        break;
    case 2:
        printf("Segunda-feira\n");
        break;
    case 3:
        printf("Terca-feira\n");
        break;
    case 4:
        printf("Quarta-feira\n");
        break;
    case 5:
        printf("Quinta-feira\n");
        break;
    case 6:
        printf("Sexta-feira\n");
        break;
    case 7:
        printf("Sabado\n");
        break;
    default:
        printf("Dia invalido\n");
        break;
}
```

Neste exemplo, se dia for igual a 3, o programa imprimirá "Terça-feira". Se o valor de dia não corresponder a nenhum case, o default será executado, indicando um "Dia inválido".

8.2 Estruturas de Repetição

8.2.1 For

A estrutura for é usada para repetir um bloco de código um número específico de vezes. É ideal para loops onde o número de iterações é conhecido previamente.

Sintaxe:

```c
for(inicializacao; condicao; incremento) {
    // Codigo a ser executado enquanto a condicao for verdadeira
}
```

Detalhes:

- Inicialização: Define a variável de controle do loop e é executada uma vez antes do início do loop.
- Condição: Verificada antes de cada iteração. Se for verdadeira, o bloco de código é executado. Se for falsa, o loop termina.

- Incremento: Atualiza a variável de controle após cada iteração.

Exemplo:

```c
#include <stdio.h>

for (int i = 0; i < 5; i++) {
    printf("i = %d\n", i);
}
```

Neste exemplo, o loop imprime os valores de i de 0 a 4. A variável i é incrementada a cada iteração até que a condição i $<$ 5 seja falsa.

8.2.2 While

A estrutura while executa um bloco de código enquanto a condição for verdadeira. A condição é verificada antes de cada iteração.

Sintaxe:

```c
while (condicao) {
    // Codigo a ser executado enquanto a condicao for verdadeira
}
```

Exemplo:

```c
#include <stdio.h>

int i = 0;
while (i < 5) {
    printf("i = %d\n", i);
    i++;
}
```

Neste exemplo, o loop imprime os valores de i de 0 a 4. A variável i é incrementada dentro do bloco de código.

8.2.3 Do while

A estrutura do while é semelhante ao while, mas a condição é verificada após a execução do bloco de código, garantindo que o bloco seja executado pelo menos uma vez.

Sintaxe:

```c
do {
    // Codigo a ser executado pelo menos uma vez e enquanto a condicao
    for verdadeira
} while (condicao);
```

Exemplo:

```c
#include <stdio.h>

int i = 0;
do {
    printf("i = %d\n", i);
    i++;
} while (i < 5);
```

No exemplo acima, o loop imprime os valores de i de 0 a 4. A variável i é incrementada dentro do bloco de código, e a condição é verificada após cada iteração.

Exemplo usando todos os conceitos:

```c
#include <stdio.h>

int main() {
    int opcao, quantidade, i;
    float preco_total = 0;
    float preco_item;
    char continuar;

    printf("Bem-vindo a loja online!\n");

    do {
        // Exibindo o menu de opcoes para o usuario
        printf("\nMenu:\n");
        printf("1. Adicionar item ao carrinho\n");
        printf("2. Ver total do carrinho\n");
        printf("3. Finalizar compra\n");
        printf("Escolha uma opcao: ");
        scanf("%d", &opcao);

        switch(opcao) {
            case 1:
                // Adicionando item ao carrinho
                printf("Digite o preco do item: R$ ");
                scanf("%f", &preco_item);
                printf("Digite a quantidade: ");
                scanf("%d", &quantidade);

                // Usando 'for' para adicionar itens multiplos ao total
                for (i = 0; i < quantidade; i++) {
                    preco_total += preco_item;
                }

                printf("Item(s) adicionado(s) ao carrinho.\n");
                break;

            case 2:
                // Exibindo o total do carrinho
                printf("Total do carrinho: R$ %.2f\n", preco_total);
                break;

            case 3:
                // Finalizando a compra
                printf("Finalizando a compra...\n");

                // Usando 'while' para verificar se o carrinho esta vazio
                while (preco_total == 0) {
                    printf("Seu carrinho esta vazio! Adicione itens antes de finalizar.\n");
                    printf("Digite 1 para adicionar itens ou 0 para sair: ");
                    scanf("%d", &opcao);
                    if (opcao == 0) {
```

```
51                          return 0;
52                      }
53                      printf("Digite o preco do item: R$ ");
54                      scanf("%f", &preco_item);
55                      printf("Digite a quantidade: ");
56                      scanf("%d", &quantidade);
57
58                      // Usando 'for' para adicionar itens multiplos ao
    total
59                      for (i = 0; i < quantidade; i++) {
60                          preco_total += preco_item;
61                      }
62                  }
63
64                  // Simulando processo de pagamento
65                  printf("Seu pagamento de R$ %.2f foi realizado com
    sucesso!\n", preco_total);
66
67                  // Usando 'do-while' para perguntar se o usuario deseja
    fazer outra compra
68                  do {
69                      printf("Deseja fazer outra compra? (s/n): ");
70                      scanf(" %c", &continuar);
71
72                      if (continuar == 's' || continuar == 'S') {
73                          preco_total = 0; // Reseta o carrinho para a
    nova compra
74                      } else if (continuar == 'n' || continuar == 'N') {
75                          printf("Obrigado por comprar conosco!\n");
76                          return 0;
77                      } else {
78                          printf("Opcao invalida! Tente novamente.\n");
79                      }
80                  } while (continuar != 'n' && continuar != 'N');
81                  break;
82
83              default:
84                  printf("Opcao invalida! Tente novamente.\n");
85                  break;
86          }
87      } while (1); // Loop infinito para continuar exibindo o menu ate que
        o usuario finalize
88
89      return 0;
90  }
```

Este código representa um sistema de compras online, onde o usuário pode dicionar itens ao carrinho repetidamente e verificar o total do carrinho a qualquer momento. Ao tentar finalizar a compra, o programa verifica se o carrinho não está vazio, e após a compra, o usuário pode optar por fazer outra compra ou sair do programa.

9 Arrays

Em C, um array é uma coleção de elementos do mesmo tipo armazenados em locais de memória contíguos. Você pode acessar os elementos do array usando índices. Vamos ver como trabalhar com vetores (arrays unidimensionais) e matrizes (arrays bidimensionais).

Abaixo, serão detalhadas algumas informações sobre arrays.

Tamanho:

O tamanho dos arrays pode ser definido de forma estática, ou dinâmica.

- Alocação Estática: O tamanho de um array em C deve ser conhecido em tempo de compilação. Não é possível alterar o tamanho de um array depois que ele foi declarado. Para arrays cujo tamanho não é conhecido até a execução, considere o uso de alocação dinâmica.
- Alocação Dinâmica: Para criar arrays cujo tamanho é determinado em tempo de execução, use funções de alocação dinâmica como malloc, calloc, e realloc. Lembre-se de liberar a memória alocada com free.

Por enquanto, será abordada apenas a alocação estática.

Vantagens/Desvantagens:

- Eficiência de Acesso: O acesso a elementos de um array é rápido e eficiente, pois os elementos são armazenados em locais de memória contíguos. Isso permite o acesso direto usando índices.
- Limitações de Memória: Em sistemas com memória limitada, grandes arrays podem consumir uma quantidade significativa de memória. Considere alocação dinâmica para flexibilidade.

Usos:

Arrays em Estruturas de Dados: Arrays são frequentemente usados em várias estruturas de dados, como listas, pilhas e filas, onde a indexação direta é vantajosa.

9.1 Vetor

Definição:

Um vetor é um array de uma única dimensão. É a estrutura de dados mais básica e simples em C.

Declaração e inicialização:

1. Declaração: Para declarar um vetor, você precisa especificar o tipo dos elementos e o tamanho do vetor.

```
    tipo nome_do_vetor[tamanho];

```

Exemplo:

```
    int numeros[5]; \\ Declara vetor de tamanho 5

```

2. Inicialização: Você pode inicializar um vetor no momento da declaração ou posteriormente. Se você não fornecer uma inicialização, os elementos do vetor serão preenchidos com valores padrão (zero para tipos numéricos).

   - Durante a definição:

```
        int numeros[5] = {1, 2, 3, 4, 5};

```

   - Posteriormente:

```
        int numeros[5];
        numeros[0] = 1;
        numeros[1] = 2;
        numeros[2] = 3;
        numeros[3] = 4;
        numeros[4] = 5;

```

Acesso aos elementos:

Os elementos do vetor são acessados usando índices. Os índices começam em 0 e vão até tamanho - 1.

```
#include <stdio.h>

int numeros[5] = {1, 2, 3, 4, 5};
printf("%d\n", numeros[2]); // Imprime 3
```

Acesso por meio de Estruturas de Repetição:

Podem ser usadas estruturas de repetição para percorrer todos elementos de um vetor:

```
#include <stdio.h>

int numeros[5] = {1, 2, 3, 4, 5};
for (int i = 0; i < 5; i++) {
    printf("%d\n", numeros[i]); // Imprime cada elemento do vetor
}
```

Inicialização por meio de Estruturas de Repetição:

```c
#include <stdio.h>

int vetor[10]; // Declara um vetor de 10 inteiros

    // Inicializa o vetor com valores de 1 a 10
    for (int i = 0; i < 10; i++) {
        vetor[i] = i + 1; // Atribui i + 1 ao elemento vetor[i]
    }

    // Imprime o vetor para verificar a inicializacao
    printf("Vetor inicializado:\n");
    for (int i = 0; i < 10; i++) {
        printf("%d ", vetor[i]);
    }
    printf("\n")
```

9.2 Matriz

Definição:

Uma matriz em C é um array de duas dimensões. Pode ser visualizada como uma tabela com linhas e colunas, onde cada elemento é acessado usando dois índices: um para a linha e outro para a coluna. Também, pode ser considerada como um conjunto de vetores.

Declaração e inicialização:

1. Declaração: Para declarar um vetor, você precisa especificar o tipo dos elementos e o tamanho do vetor.

```c
    tipo nome_da_matriz[num_linhas][num_colunas];

```

Exemplo:

```c
    int matriz[3][4]; // Declara uma matriz com 3 linhas e 4 colunas

```

2. Inicialização: Assim como vetores, matrizes podem ser inicializadas no momento da declaração.

   - Durante a definição:

```c
        int matriz[3][4] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
        };

```

   - Posteriormente:

```c
        int matriz[3][4]; // Declara uma matriz de 3x4

        // Inicializa a matriz com valores especificos
```

```
4           matriz[0][0] = 1;
5           matriz[0][1] = 2;
6           matriz[0][2] = 3;
7           matriz[0][3] = 4;
8           matriz[1][0] = 5;
9           matriz[1][1] = 6;
10          matriz[1][2] = 7;
11          matriz[1][3] = 8;
12          matriz[2][0] = 9;
13          matriz[2][1] = 10;
14          matriz[2][2] = 11;
15          matriz[2][3] = 12;
16
```

Acesso aos elementos:

Os elementos da matriz são acessados usando dois índices: o primeiro para a linha e o segundo para a coluna. Os índices começam em 0.

```
1 #include <stdio.h>
2
3 int matriz[3][4] = {
4     {1, 2, 3, 4},
5     {5, 6, 7, 8},
6     {9, 10, 11, 12}
7 };
8 printf("%d\n", matriz[1][2]); // Imprime 7 (linha 1, coluna 2)
```

Acesso por meio de Estruturas de Repetição:

Podem ser usadas estruturas de repetição para percorrer todos elementos de uma matriz:

```
1 #include <stdio.h>
2
3 int matriz[3][4] = {
4     {1, 2, 3, 4},
5     {5, 6, 7, 8},
6     {9, 10, 11, 12}
7 };
8
9 for (int i = 0; i < 3; i++) {
10     for (int j = 0; j < 4; j++) {
11         printf("%d ", matriz[i][j]); // Imprime cada elemento da matriz
12     }
13     printf("\n"); // Nova linha apos cada linha da matriz
14 }
```

Inicialização por meio de Estruturas de Repetição:

```
1 #include <stdio.h>
2
3 int matriz[3][4]; // Declara uma matriz de 3x4
4
5     // Inicializa a matriz com valores
6     for (int i = 0; i < 3; i++) {
7         for (int j = 0; j < 4; j++) {
8             matriz[i][j] = i * 4 + j + 1; // Exemplo de atribuicao:
    valores de 1 a 12
```

```
9           }
10      }
11
12      // Imprime a matriz para verificar a inicializacao
13      for (int i = 0; i < 3; i++) {
14          for (int j = 0; j < 4; j++) {
15              printf("%d ", matriz[i][j]);
16          }
17          printf("\n");
18      }
```

10 Recursão

Recursão é uma técnica que permite uma função chamar a si mesma, com o intuito de dividir uma tarefa em subtarefas, facilitando-se a execução do algoritmo e evitando uso de laços e repetições.

A recursão baseia-se no princípio da indução finita (PIF), no qual há um caso base para um valor inicial e uma equação ou algoritmo para obter o resultado dos outros valores, a partir do caso base.

Por exemplo, a função abaixo tem o objetivo de somar todos os números inteiros de um intervalo de 0 a um número inteiro escolhido e mostrar o resultado:

```
1
2  int soma(int k);
3
4  void main() {
5    int resultado = soma(10);
6    printf("%d", resultado);
7    // Resultado = 55
8  }
9
10 int soma(int k) {
11   if (k > 0) {
12     return k + soma(k - 1);
13   } else {
14     return 0;
15   }
16 }
```

No exemplo mostrado, o caso base da função é quando o parâmetro é 0, no qual a recursão retorna o valor 0 para a última chamada recursiva.

$10 + soma(9)$

$10 + (9 + soma(8))$

$10 + (9 + (8 + soma(7)))$

...

$10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + soma(0)10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0$

Note-se também que a soma de todos os números é feita somando dois números de cada vez no algoritmo.

10.1 Registro de ativação (Adendo)

Para entender mais a fundo como funciona a recursão nas liguagens de programação deve-se entender o conceito de registro de ativação. Registro de ativação é uma referência a uma função que foi inicializada e está sendo processada. Neste sentido, quando uma função A chama uma função B, seja recursivamente ou não,pausa-se a execução de A até que a execução de B seja finalizada, portanto ativa-se B e cria-se um registro de ativação de B dentro de A, para que o resultado de B retorne para A.

Caso B chame uma função C, ocorre-se o mesmo processo. Desta forma, para que o computador não se confunda qual função executar em uma recursão, os registros de ativação são organizados na forma de pilha, descrita como pilha de ativação.

Neste sentido, a cada método que uma recursão chama, o registro de ativação do método chamado é posto no topo da pilha e inicia-se a execução deste método. Quando chega-se ao caso base da recursão, a função do caso base é executada e seu registro de ativação que está no topo da pilha é retirado, e então o registro de ativação da função anterior torna-se o topo da pilha e executa-se a função anterior, retirando-se também seu registro da pilha após a execução . Logo serão sempre executadas as funções nas quais seus registros de ativação estão no topo da pilha e finaliza-se o algoritmo até que todos os registros de ativação sejam retiradas da pilha.

11 Ponteiro

11.1 Endereço de memória

Para entender ponteiro em C é fundamental conhecer sobre o conceito de endereçamento de memória. Resumidamente, cada operando e instrução em uma algoritmo é guardado na memória do computador em uma célula(espaço) específico e cada célula possui um endereço para que o computador possa encontra-la e utilizar a informação que está sendo armazenada.

Exemplo:

Endereço de memória	Célula de memória
*100	a = 5.
*101	b = 8.
*102	a + b.

Tabela 3: Exemplo de armazenamento de instruções e variáveis em memória

11.2 Ponteiros: conceito e declaração

11.2.1 Conceito

O ponteiro em C tem a ação de apontar para um endereço de memória, que pode estar armazenando um valor e uma variável.

Exemplo:

Endereço de memória	Célula de memória
*100	a = 5.
*101	b = 8.
*102	a + b.
*103	*c -> *100 (a).

Tabela 4: Exemplo do funcionamento de ponteiro

11.2.2 Aplicação

A declaração de um ponteiro e suas propriedades é demonstrado neste exemplo abaixo:

```
1
2 int minhaIdade = 19;      // Variavel
3 int* A = &minhaIdade;  // Declaracao do ponteiro A que aponta para o
     endereco de memoria da variavel minhaIdade.
4
5 // Resultado da variavel minhaIdade (19)
6 printf("%d\n", minhaIdade);
7
8 // Resultado do endereco de memoria de minhaIdade (0x7ffe5367e044)
9 printf("%p\n", &minhaIdade);
10
11 // Resultado do endereco de memoria de minhaIdade atraves do ponteiro A
     (0x7ffe5367e044)
```

```
12 printf("%p\n", A);
13
14 // Valor da variavel minhaIdade atraves do ponteiro A (19)
15 printf("%d\n", *A);
```

Como visto, para se declarar um ponteiro é necessário declarar o tipo de dado que ele aponta e colocar um asterisco(*) antes de seu nome.

Se a variável apontada pelo ponteiro tiver seu valor alterado, o endereço de memória referenciado pelo ponteiro não muda, mas o valor armazenado no endereço sim.

Exemplo:

```
1
2 int minhaIdade = 19;
3 int* A = &minhaIdade;
4
5 // Resultado do endereco de memoria de minhaIdade atraves do ponteiro A
     (0x7ffe5367e044)
6 printf("%p\n", A);
7
8 // Valor da variavel minhaIdade atraves do ponteiro A (19)
9 printf("%d\n", *A);
10
11 // Valor de minhaIdade alterado
12 minhaIdade = 30;
13
14 // Resultado do endereco de memoria de minhaIdade atraves do ponteiro A
     (0x7ffe5367e044)
15 printf("%p\n", A);
16
17 // Valor da variavel minhaIdade atraves do ponteiro A (30)
18 printf("%d\n", *A);
```

Outras propriedades dos ponteiro é que mais de um ponteiro pode apontar para a mesma variável, e que um ponteiro pode apontar para o outro.

Exemplo:

```
1
2 int minhaIdade = 19;
3 int* A = &minhaIdade;
4 int* B = &minhaIdade;
5 int** C = &B;
```

Também é possível realizar operações aritméticas com ponteiros

Exemplo:

```
1 #include <stdio.h>
2 #include <string.h>
3 void main() {
4
5     int x = 4;
6     int *p = &x;
7     *p *= 2;
8     printf("\n %d", x); // x = 8
9     printf("\n %d", *p); // *p = 8
10
11    int y = 3 * *p;
12    printf("\n %d", y); // y = 24
```

```
13
14      char a[] = {"World"};
15      char* b = &a;
16      char c[] = {"Hello"};
17
18
19      printf("\n%s %s",c, b); // Hello World
20
21
22  }
```

Quando um ponteiro aponta para um array, o endereço de memória que o ponteiro aponta é equivalente ao endereço de memória do primeiro elemento do array. Além disso é possível incrementar ou decrementar o endereço de memória do ponteiro para ele apontar para outras variáveis do vetor.

Exemplo:

```
1  void main(){
2
3  int numeros[4] = {25, 50, 75, 100};
4  int *ponteiro = &numeros; //Ponteiro apontando para o array.
5
6  // Endere o de mem ria apontado pelo ponteiro (0x0061FF0C)
7  printf("%p\n", ponteiro);
8
9  // Valor armazenado no endere o de mem ria apontado (25)
10 // Perceba que 25    o primeiro elemento do array numeros
11 printf("%d\n", *ponteiro);
12
13 // Faz o ponteiro apontar para a c lula de mem ria seguinte,
14 // que neste caso    o segundo elemento do array
15 ponteiro += 1;
16
17 // Endere o de mem ria do apontado pelo ponteiro (0x0061FF10)
18 printf("%p\n", ponteiro);
19 // Valor armazenado no endere o de mem ria apontado (50)
20 printf("%d\n", *ponteiro);
```

Como uma string no C é um vetor de chars, a lógica dos ponteiro com strings é quase em relação a vetores

Exemplo

```
1  void main(){
2
3      char x[] = {"Hello World"};
4      char* y = x;
5      // *y -> 'H'
6      printf("\n%s", y); // Hello World
7      y += 2;
8      // *y -> 'l'
9      printf("\n%s", y); // llo World
10     y -=1;
11     // *y -> 'e'
12     printf("\n%s", y); // ello World
13
14
15 }
```

12 Função com Passagem de Parâmetros por Valor e por Referência

12.1 Função com passagem por Valor

Na função de passagem por valor,o parâmetro da função recebe uma variável que está fora da função e a função produz uma cópia desta váriavel dentro dela. Se a função modificar o valor da variável, a variável será alterada apenas dentro da função e não fora.

Exemplo:

```c

#include <stdio.h>

void multiplicadordois(int A){

    A = A * 2;
    printf("\nResultado: %d", A);
    printf("\nEndereco de memoria: %X", &A);
}
void main()
{
    int A = 10;

    multiplicadordois(A);
    // Resultado: 20
    // Endereco de memoria: 61FF00

    printf("\nResultado %d",A);
    //Resultado: 10
    printf("\nEndereco de memoria : %X", &A);
    // Endereco de memoria: 61FF18

}
```

Como pode-se notar no exemplo acima, o valor da váriavel A é alterado dentro da função multiplicadordois(), mas não fora.

12.2 Função com passagem por Referência

Na função por passagem por referência, a função recebe o endereço de memória de uma variável como parâmetro. Neste caso, se o valor referenciado pelo parâmetro for alterado pela função, então o valor da variável que está fora da função também é alterado.

Exemplo:

```c

#include <stdio.h>

void multiplicadoDoisV2(int *A){
    *A = *A * 2;
    printf("\n Resultado: %d", *A);
    printf("\nEndereco de memoria referenciado: %X", A);
}

```

```
10
11 void main(){
12     int A = 10;
13
14     multiplicadoDoisV2(&A);
15     // Reultado: 20
16     // Endereco de memoria referenciado: 61FF1C
17
18     printf("\n Resultado: %d", A);
19     // Resultado: 20
20     printf("\nEndereco de memoria: %X ", &vA);
21     // Endereco de memoria: 61FF1C
22 }
```

Como pode-se notar no exemplo acima, o valor da variável A é alterado dentro e fora da função multiplicadordois(), pois a função utiliza o mesmo endereço de memória de A para realizar suas operações.

Note-se também que na função multiplicadordois() foi utilizado ponteiros para armazenar o valor do endereço de memória de seu parâmetro.

13 Extra

Neste workshop foi passado a sintaxe básica da linguagem C, porém algumas outras propriedades da linguagem não foram mostradas por falta de tempo e por serem mais complexas, principalmente para quem está iniciando na linguagem.

a seguir há alguns tópicos da linguagem que não foram vistos aqui, mas que você pode estudar e se aprofundar depois.

- Estrutura (struct)
- Enumeração(enum)
- Alocação de memória(calloc, malloc, realloc, free)
- Arquivos(fopen, fclose)
- Manipulação de bits(«, »)

Links para de materiais para estudar as outras propriedades

- https://www.w3schools.com/c/index.php
- https://embarcados.com.br/bits-em-linguagem-c/

Referências

[1] Scott Chacon and Ben Straub. *Pro Git*. Apress, Berkeley, CA, 2nd edition, 2014.

[2] International Organization for Standardization. *ISO/IEC 9899:2018: Information technology – Programming languages – C*, 5th edition, 2018.

[3] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1978.

[4] Dennis M. Ritchie. The development of the c language. *ACM SIGPLAN Notices*, 28(3):201–208, 1993.

[5] Robert W. Sebesta. *Concepts of Programming Languages*. Pearson, 10th edition, 2012.