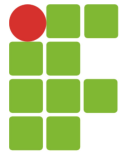
Programação Avançada

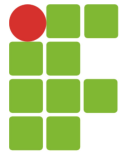
Programação de Computadores I – Ciência da Computação

Prof. Daniel Saad Nogueira Nunes

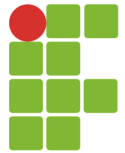
IFB – Instituto Federal de Brasília
Campus Taguatinga



Sumário

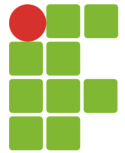


Sumário

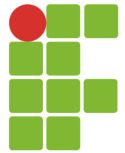


Introdução

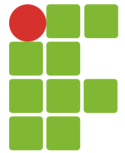
- Neste capítulo abordaremos vários conceitos importantes da programação C.
- Especificamente, trataremos de:
  - Depuração;
  - Modularização de código;
  - Testes Unitários;
  - Makefiles e CMake.



Sumário



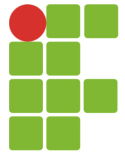
Sumário



Introdução

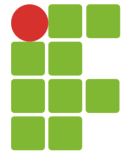
De acordo com A. Miseravi:

Programar é uma arte, depurar faz parte.



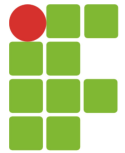
Depuração

- É inevitável que os programas produzidos apresentem *bugs*, ou falhas de lógica
- Quanto maior o programa, maior a probabilidade de erros.
- A depuração consiste em encontrar e eliminar defeitos em softwares.
- Podemos usar depuradores (debuggers) para auxiliar neste processo.



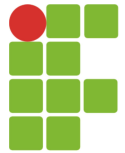
Depuração

Figura: Usar printf para achar o erro. Pode sim amiguinho!



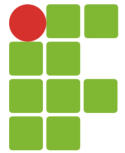
Depuração

- Brincadeiras a parte, utilizar comandos de impressão em telas ou em arquivos para checar variáveis é um método de depuração denominado *Print debugging* (ou *tracing*).
- Nem sempre é efetivo.

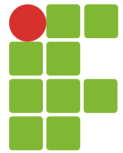


Depuração

- Alguns depuradores possuem uma série de ferramentas que ajudam a localizar erros, tais como:
  - Parar a execução em determinadas linhas de códigos (ou funções).
  - Pular chamadas de funções e continuar a análise após a sua chamada.
  - Imprimir o conteúdo das variáveis. em determinada linha de código.
- Estes depuradores se encontram frequentemente atrelados à IDEs ou via linha de comando.
- Aprenderemos a utilizar o `GDB`.

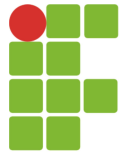


Sumário



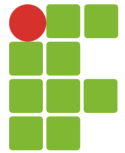
GDB

- Para utilizar o `GDB`, é necessário compilar os códigos fonte com a flag `-g`
- `gcc -g arquivo.c -o <nome_executável>`



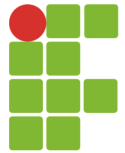
GDB

- Primeiramente, é necessário carregar o executável atravé do GDB.
  - `gdb ./<nome_do_executável>`



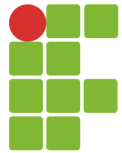
GDB

- Breakpoints, são pontos de parada.
- Toda vez que o depurador atinge uma linha do código marcada com breakpoint, ele para de rodar.
- É interessante colocar breakpoints em pontos problemáticos do código para detectar as falhas de lógica.
- Sintaxe:
  - `break <número_da_linha>`.
  - `break <nome_da_função>`.
- O comando `tbreak` possui a mesma sintaxe, mas assim que o programa atinge o este ponto temporário, ele é removido.



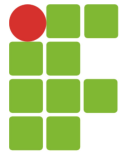
GDB

- Para dar início à execução do programa, utilizamos:
  - `run`
- O programa rodará até o final, ou até atingir o breakpoint mais próximo.



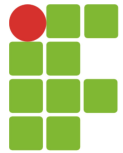
Comandos GDB

- `clear`: deleta todos os breakpoints.
- `clear <nome_da_função>`: deleta todos os breakpoints da função.
- `clear <número_da_linha>`: deleta os breakpoints relativos à linha especificada.



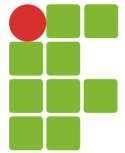
Comandos GDB

- `continue`: continue executando até atingir o próximo breakpoint ou até o término do programa.
- `step`: executa a próxima instrução. Caso seja uma função, entra na função.
- `step n`: performa n `steps`.
- `s`: abreviação de step.
- `next`: executa a próxima instrução. Caso seja uma função, executa a função e pula para a próxima instrução.
- `next n`: performa n `next`.
- `n`: abreviação de next.



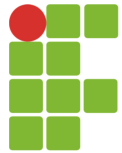
Comandos GDB

- `until <nome_da_funcao>`: continua a execução até atingir o nome da função.
- `until <número_da_linha>`: continua a execução até atingir o número de linha especificado.



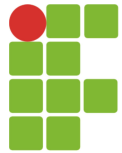
Comandos GDB

- `where`: mostra o número da linha corrente e o nome da função que está sendo executada no momento.
- `backtrace`: Imprime as funções empilhadas.



Comandos GDB

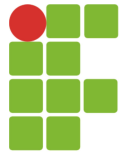
- teste.
- `list`: imprime o código fonte.
- `list <nome\_da\_função>`: imprime o código fonte a partir da função especificada.
- `list <número\_de\_linha>`: imprime o código fonte a partir do número da linha especificado.
- `list <start>, <end>`: imprime da linha `start` até a linha `end` do código fonte.



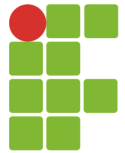
Comandos GDB

- `print <nome_da_variável>`: imprime o valor da variável.
- `print <nome_da_variável>`: imprime o valor da variável.
- `print *<vetor>@<tamanho>`: imprime `tamanho` valores do `vetor`.
- `p`: abreviação de `print`.
- `p/x <nome_da_variável>`: imprime a variável em hexadecimal.
- `p/d <nome_da_variável>`: imprime a variável como inteiro com sinal.
- `p/u <nome_da_variável>`: imprime a variável como inteiro sem sinal.
- `p/o <nome_da_variável>`: imprime a variável como octal.

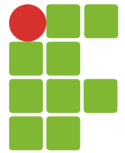
Depuração



Sumário

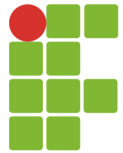


Sumário



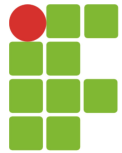
Modularização

- O código da linguagem C pode ser quebrado em diversas partes.
- Cada função é responsável por resolver um pedaço da tarefa.
- Isola erros e facilita o entendimento do código, bem como o desenvolvimento.
- Permite reuso de código com mais facilidade.



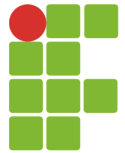
Modularização

- A medida que o sistema cresce, é conveniente separá-lo em diversos arquivos fonte (módulos).
- Cada arquivo fonte é responsável pela implementação de uma funcionalidade do sistema.
- Os arquivos fontes são compilados para criar um único executável.
- Cada arquivo pode ser classificado em:
  - Cabeçalho (arquivos .h).
  - Implementação (arquivos .c).
- Isto isola e encapsula os dados de cada arquivo.

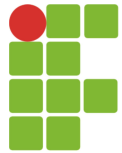


Modularização

- Encapsulamento é um mecanismo de linguagem de programação para esconder os dados e evitar acesso direto aos mesmos.
- Desta forma, os dados são acessíveis e manipulados com funções, exportadas por cada módulo do sistema.

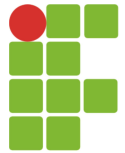


Sumário

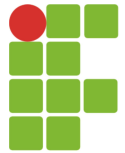


Arquivos Cabeçalho

- São arquivos com extensão .h (header).
- Possuem o contrato daquela funcionalidade.
- Este contrato consiste dos protótipos das funções que devem ser utilizadas pelos outros módulos.
- Para utilizar estas funções, deve-se incluir o arquivo cabeçalho através da diretiva `#include`.



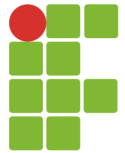
Sumário



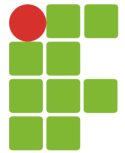
Arquivos de Implementação

- Possui a implementação do contrato.
- Devem implementar os protótipos das funções declaradas no arquivo cabeçalho.
- Para compilar um arquivo de implementação, utilizamos:
  - `gcc -c <nome_do_arquivo.c>`.
- Isto gerará um arquivo objeto chamado `nome_do_arquivo.o`.
- Os arquivos objetos dos arquivos de implementação são ligados (linked) em um único executável.

Modularização

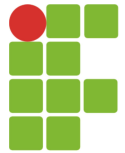


Sumário



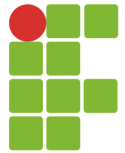
Bibliotecas

- Bibliotecas são uma coleção de rotinas, funções e variáveis que fornecem alguma funcionalidade ao sistema.
- Elas facilitam a reutilização de código em outros programas.
- Podem ser:
  - Estáticas.
  - Dinâmicas.



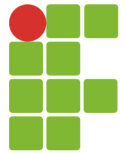
Bibliotecas

- Bibliotecas estáticas: são ligadas juntamente com outros arquivos objetos para geração do executável.
  - No Linux: arquivos com extensão **.a**.
- Bibliotecas dinâmicas (shared): são carregadas em tempo de execução devido ao carregador (loader).
  - No Linux: arquivos com extensão **.so**.



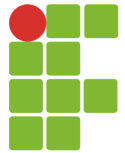
Bibliotecas Dinâmicas

- Para gerar biblioteca dinâmica com nome `libclass.so` dos objetos `class1.o` `class2.o` e `class3.o`:
  - `gcc -shared -o libclass.so class1.o class2.o class3.o`
- Para ligar os objetos e formar o executável propriamente dito, precisamos informar o caminho de onde está a biblioteca e indicar o nome da mesma (abreviado):
  - `gcc -L<caminho_da_biblioteca> main.c -o main -lclass`



Bibliotecas Dinâmicas

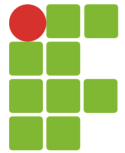
- Para gerar biblioteca dinâmica com nome `libclass.so` dos objetos `class1.o` `class2.o` e `class3.o`:
  - `gcc -shared -o libclass.so class1.o class2.o class3.o`
- Para ligar os objetos e formar o executável propriamente dito, precisamos informar o caminho de onde está a biblioteca e indicar o nome da mesma (abreviado):
  - `gcc -L<caminho_da_biblioteca> main.c -o main -lclass`



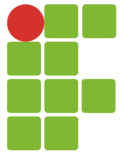
Bibliotecas Dinâmicas

- Se a biblioteca dinâmica não encontra-se em pastas do sistema, é necessário especificar o caminho de onde se encontra via a variável de ambiente `LD_LIBRARY_PATH` da seguinte forma:

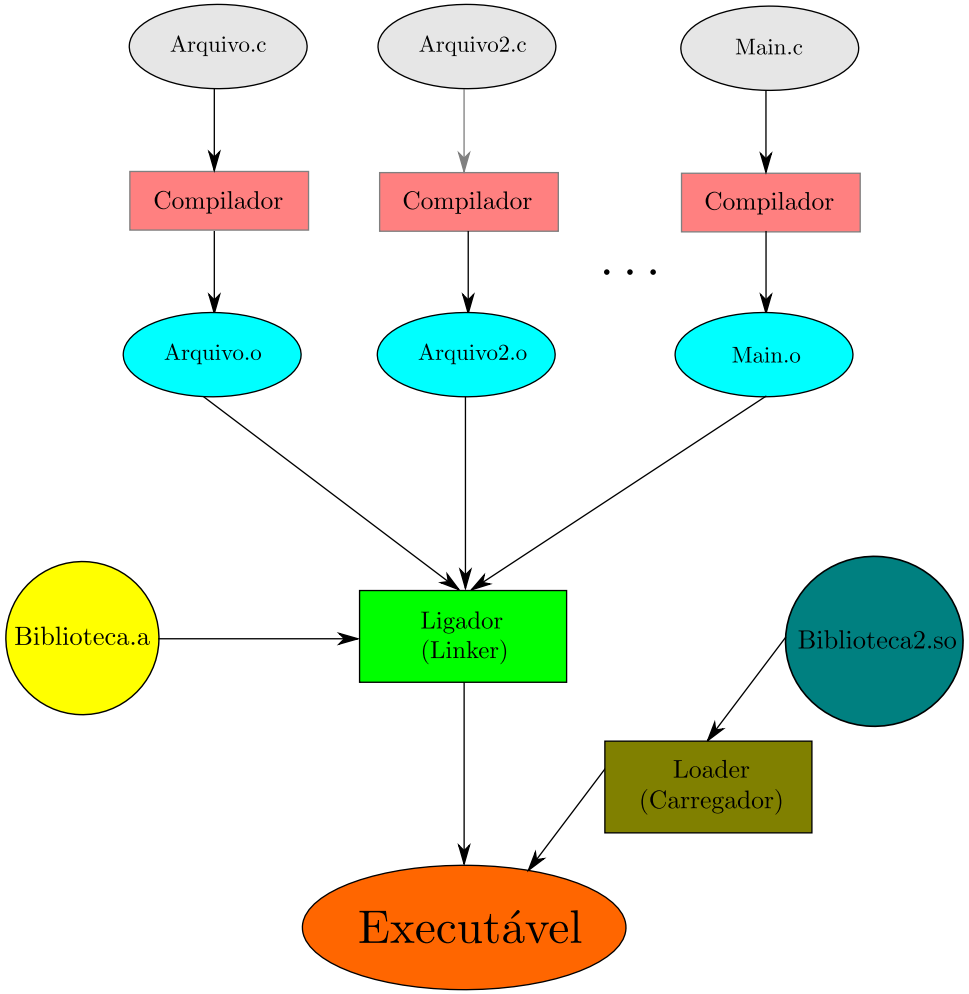
```
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:<CAMINHO DA BIBLIOTECA>
export LD_LIBRARY_PATH
```

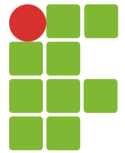


Sumário

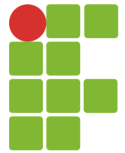


Compilação e Ligação

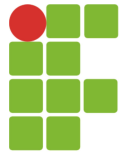




Sumário

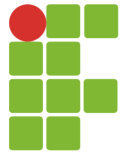


Sumário

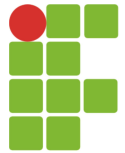


Testes Unitários

- O teste de unidade consiste em verificar se uma unidade (menor parte testável de um programa) apresenta o comportamento correto.
- No caso de linguagens procedurais, as unidades comumente são as funções.
- É imprescindível. Assegura que o software apresenta o design e implementação correta.
- Reduz custos! Quanto mais cedo testes são feitos, mais cedo os problemas são encontrados e menos tempo é gasto em manutenção.
- Use testes unitários!

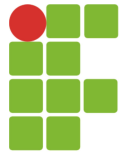


Sumário



O Framework Check

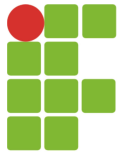
- O Framework Check é um framework de testes de unidade para a linguagem C.
- `https://libcheck.github.io/check/`



O Framework Check

Conceitos Básicos

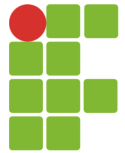
- **Casos de teste:** os testes de unidade própriamente ditos.
- **Suites:** coleção de casos de teste.
- **Runners**: executam as suítes de teste.



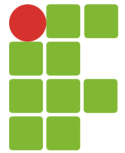
Testes Unitários

- O Framework Check exibe os testes que passaram e quais falharam após rodar as suites.
- Todos os testes devem passar.
- Em cada caso de testes, utilizamos a macro `ck_assert` sob expressões. Caso esta macro seja avaliada em falso, o teste falha automaticamente.
- Vamos ver isso em código?

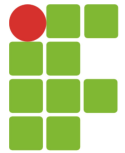
Framework Check



Sumário

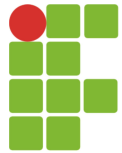


Sumário



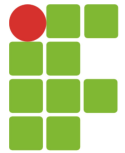
Makefiles

- **Makefiles** indicam como a ferramenta `make` deve compilar e ligar arquivos fontes de um sistema.
- De acordo com o grafo de dependências, ao alterar um arquivo, a compilação é aplicada apenas nos arquivos necessários.
- Reduz o tempo de compilação.
- Essencial em projetos maiores.



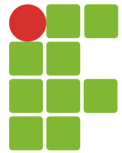
Exemplo

```
edit: main.o kbd.o command.o display.o
    gcc -o edit main.o kbd.o command.o display.o
main.o: main.c defs.h
    gcc -c main.c
kbd.o : kbd.c defs.h command.h
    gcc -c kbd.c
command.o: command.c defs.h command.h
    gcc -c command.c
display.o : display.c defs.h
    gcc -c display.c
clean:
    rm edit main.o kbd.o command.o display.o
```



Makefiles

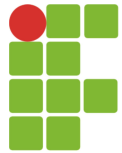
- Para executar o makefile, utilize o terminal na pasta onde se encontra o makefile e digite:
  - `make`.
- Alguns makefiles possuem uma regra de instalação, que move arquivos para pastas corretas, geralmente esta regra executada com o comando:
  - `make install`.



Makefiles

Opinião do Professor

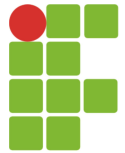
Makefiles possuem uma sintaxe muito confusa na minha opinião. Fazer coisas mais complexas com ele dá mais trabalho, além de ser complicado criá-los de maneira multi-plataforma. Eu prefiro o CMake.



CMake

O que é o CMake?

- Diferentemente dos Makefiles, não gera o sistema final.
- Cria arquivos de geração automatizada (Ex: makefiles).
- Processo dividido em duas partes: cria-se os arquivos de geração automatizada e depois, utilizamos ferramentas nativas do sistema operacional para criar o software (Ex: `make`).



Sumário

Utilizando o CMake