E07 — Compilação Separada em C

Disciplina de Compiladores

9 de novembro de 2025

Revisão

Na aula, vimos como separar um programa em C em múltiplos arquivos, distinguindo **protótipos** (arquivos `.h`) de **implementações** (arquivos `.c`). Essa prática traz vantagens importantes:

- **Modularização**: cada funcionalidade fica em seu próprio módulo
- **Recompilação eficiente**: modificar um arquivo não requer recompilar tudo
- **Reusabilidade**: módulos podem ser reutilizados em outros projetos
- **Organização**: código fica mais limpo e fácil de manter

O processo de compilação envolve três etapas principais:

1. **Pré-processamento**: comandos com `#` são processados (ex.: `#include`)
2. **Compilação**: cada `.c` é compilado em um arquivo objeto `.o`
3. **Linkagem**: todos os `.o` são ligados para gerar o executável final

Repositório de Referência

Para este exercício, utilizaremos como ponto de partida o projeto disponível em:

https://github.com/tioguerra/ComilaSeparadoC

Clone o repositório com:

```
git clone https://github.com/tioguerra/ComilaSeparadoC.git
cd ComilaSeparadoC
```

Explore o código e compile com `make` para entender como funciona. Você notará que o `Makefile` deste projeto é um pouco diferente do que vimos em aula. Vamos entender essas diferenças.

Makefile

O `Makefile` do repositório usa **variáveis**, um conceito que não exploramos em aula. Vejamos um exemplo:

```
CC := gcc
TARGET := main
OBJS := main.o soma.o subtrai.o
```

Aqui estamos **definindo** três variáveis:

- `CC` recebe o valor `gcc`
- `TARGET` recebe o valor `main`
- `OBJS` recebe o valor `main.o soma.o subtrai.o`

Mais adiante no Makefile, quando quisermos **usar** essas variáveis, escrevemos `$(nome_da_variável)`. Por exemplo:

```
$(TARGET): $(OBJS)
    $(CC) $(OBJS) -o $(TARGET)
```

O `make` vai **substituir** cada variável pelo seu valor. Então essa regra é equivalente a escrever:

```
main: main.o soma.o subtrai.o
    gcc main.o soma.o subtrai.o -o main
```

Perceba que `-o` aqui é uma opção (também chamada de flag) que diz ao compilador: "o próximo argumento é o nome do arquivo de saída". Então `-o main` significa "crie um executável chamado main".

Flags

Além de `-o`, existem muitas outras flags que modificam como o compilador funciona. No projeto de referência, usamos:

```
CFLAGS := -Wall -Wextra -pedantic -std=c11
```

O que cada uma faz?

- `-Wall` e `-Wextra`: ativam **avisos** (warnings). O compilador te alerta sobre código suspeito, mesmo que não seja erro. Por exemplo: "você declarou uma variável mas nunca usou", ou "essa comparação sempre será verdadeira". São seus aliados!
- `-pedantic`: força o código a seguir rigorosamente o padrão C, sem usar extensões específicas do GCC. Isso torna seu código mais portável.
- `-std=c11`: especifica qual versão do padrão C você quer usar. No caso, C11 (de 2011). Sem essa flag, o compilador pode usar um padrão mais antigo ou uma versão própria.

Essas flags ajudam você a escrever código mais correto e profissional. Warnings hoje evitam bugs amanhã!

Outras Variáveis Úteis

Vejamos as demais variáveis do Makefile:

- `SRCS := main.c soma.c subtrai.c` — Lista todos os arquivos fonte (`.c`) do projeto.
- `OBJS := $(SRCS:.c=.o)` — Isso é uma **substituição automática**. Ela pega cada nome em `SRCS` e troca `.c` por `.o`. Assim, não precisamos escrever a lista de `.o` manualmente! Se `SRCS` contém `main.c soma.c subtrai.c`, então `OBJS` automaticamente terá `main.o soma.o subtrai.o`.
- `.PHONY: all clean run` — Declara que `all`, `clean` e `run` são **comandos**, não arquivos. Isso evita problemas caso exista um arquivo com esses nomes.
- `$@` — É uma variável **automática** do `make` que representa o nome do alvo (nome antes dos dois pontos) da regra atual. Na regra:

```
$(TARGET): $(OBJS)
    $(CC) $(OBJS) -o $@
```

  O `$@` será substituído por `main` (valor de `TARGET`), resultando em: `gcc main.o soma.o subtrai.o -o main`.

Por que usar variáveis? Imagine que você adicione um novo módulo. Sem variáveis, teria que modificar várias linhas do Makefile. Com variáveis, basta adicionar o nome do arquivo em `SRCS` e tudo mais funciona automaticamente!

Tarefa (em duplas)

O que fazer

Você irá **estender significativamente** o projeto de exemplo. O projeto atual implementa apenas duas operações (soma e subtração). Sua tarefa é criar **pelo menos quatro novos módulos**, cada um com uma operação aritmética diferente, e integrá-los ao programa.

Ao final, o programa deverá executar e imprimir **pelo menos seis operações**: soma, subtração e as quatro (ou mais) novas operações que você implementar.

Operações sugeridas

Você deve escolher pelo menos quatro das seguintes operações (ou criar outras similares):

- Multiplicação: `int multiplica(int a, int b);`
- Divisão (inteira): `int divisao(int a, int b);`
- Módulo (resto): `int modulo(int a, int b);`
- Potência: `int potencia(int base, int exp);`
- Máximo: `int maximo(int a, int b);`

- Mínimo: `int minimo(int a, int b);`
- Valor absoluto: `int absoluto(int x);`
- Fatorial: `int fatorial(int n);`

Passo a passo

1. **Clone e explore o repositório**:

```
git clone https://github.com/tioguerra/ComilaSeparadoC.git
cd ComilaSeparadoC
make
make run
```

   Estude a estrutura: veja como `soma.h` e `soma.c` estão organizados, como `main.c` os utiliza, e como o `Makefile` orquestra tudo.

2. **Para cada novo módulo que você criar**:

   a) Crie o arquivo `.h` (header):

   - Coloque apenas o **protótipo** da função
   - Use `#IFNDEF ... #ENDIF` (veja `soma.h` como exemplo)

   b) Crie o arquivo `.c` (implementação):

   - Na primeira linha, inclua o respectivo `.h`
   - Implemente a função

3. **Modifique o `main.c`**:

   - Adicione um `#include` para cada novo header
   - **Importante**: nunca inclua arquivos `.c`! Sempre `.h`.
   - Chame cada uma das novas funções e imprima os resultados
   - Teste com valores que façam sentido para cada operação

4. **Atualize o `Makefile`**:

   a) Na linha que define `SRCS`, adicione todos os novos arquivos `.c`.

   b) Para cada novo módulo, adicione uma regra de dependência (similar às que já existem para `main.o`, `soma.o` e `subtrai.o`). A regra tem este formato:

```
nome.o: nome.c nome.h
```

   Isso diz ao `make`: "se `nome.c` ou `nome.h` mudarem, recompile `nome.o`". Adicione uma linha dessas para cada módulo que você criou.

   Atenção: não precisa escrever o comando de compilação (`gcc -c ...`) porque o `make` tem regras implícitas para isso. Apenas declare as dependências!

5. **Teste extensivamente**:

```
make clean     # Remove arquivos antigos
make           # Compila tudo do zero
make run       # Executa o programa
```

Verifique:

- Não há erros de compilação
- Não há warnings (o compilador não deve reclamar de nada)
- Todos os resultados são impressos corretamente
- Os cálculos estão corretos

6. **Teste modificações**:

   Edite um dos seus arquivos `.c`, mude algo simples (como um valor de retorno), salve e rode `make` novamente. Observe que apenas aquele `.o` é recompilado, não tudo! Isso demonstra a eficiência da compilação separada.

Entrega

Crie um **novo repositório público** no GitHub (não faça fork) contendo:

- Todos os arquivos `.c` e `.h`:
  - Os originais: `main.c`, `soma.c/.h`, `subtrai.c/.h`
  - Seus novos módulos (pelo menos 4)
- O `Makefile` atualizado com:
  - Variável `SRCS` incluindo todos os `.c`
  - Regras de dependência para todos os novos `.o`
- Um `README.md` bem escrito contendo:
  - Breve descrição do projeto
  - Lista das operações implementadas
  - Como compilar: `make`
  - Como executar: `make run` ou `./main`
  - Exemplo de saída completa mostrando todos os resultados
  - Como limpar: `make clean`
- (Opcional mas fortemente recomendado) Um arquivo `.gitignore` contendo:

```
*.o
main
```

  Isso evita commitar arquivos compilados no repositório.

Submeta o link do repositório no ambiente da disciplina.

Avaliação (checklist)

- ☐ Pelo menos 4 novos módulos criados (além de soma e subtrai).
- ☐ Cada módulo tem separação correta: `.h` com protótipo, `.c` com implementação.
- ☐ Include guards corretos em todos os `.h`.
- ☐ `main.c` modificado para usar todas as funções e imprimir todos os resultados.
- ☐ `main.c` inclui apenas headers, nunca arquivos `.c`.
- ☐ `Makefile` atualizado: variável `SRCS` completa e regras de dependência para todos os novos `.o`.
- ☐ Compilação bem-sucedida sem erros.
- ☐ Compilação sem warnings (graças a `-Wall -Wextra`).
- ☐ Executável funciona corretamente e imprime todos os resultados.
- ☐ `README.md` claro, completo e bem formatado.
- ☐ Repositório público no GitHub com estrutura limpa.