Modularização em C

Prof. Marcelo Veiga Neves

Motivação: por que modularizar?

- O que acontece se o programa é composto por muitas funções?
  - Todas ficam dentro do mesmo código-fonte
  - Código pode se tornar muito extenso e difícil de gerenciar
- Isso também cria dois outros problemas:
  - Dificulta a reutilização das funções em outros programas
  - Dificulta o trabalho em equipe
- Então, qual é a solução?

Solução: Modularização

- Agrupam-se as funções relacionadas em módulos. Para cada módulo são utilizados dois arquivos:
- Declarações (arquivos .h)
  - Contém apenas as declarações das funções (protótipos) e das constantes
- Implementação (arquivos .c)
  - Contém a implementação das funções declaradas no arquivo .h correspondente
  - Inclui o arquivo .h correspondente
- Para usar uma função em outro módulo, basta então incluir o arquivo .h correspondente

Criação do header da biblioteca de funções (1)

- Para exemplificar, criaremos uma biblioteca simples de funções matemáticas
- Primeiro, escrevemos os protótipos das funções no arquivo bibfunc.h:

```
int fatorial(int v);
int somatorio(int v);
...
```

Criação do header da biblioteca de funções (2)

- Porém, se por engano esse arquivo for incluído mais de uma vez, haverá uma duplicidade de todas as definições
- Como isso pode acontecer facilmente, acrescentamos 3 linhas no arquivo .h:

```
#ifndef BIBFUNC_H
#define BIBFUNC_H

int fatorial(int v);
int somatorio(int v);
...
#endif
```

- Essas linhas contêm comandos do pré-processador, que definem um símbolo BIBFUNC_H caso este ainda não exista
- Se existir, é porque o arquivo já foi incluído e portanto, o pré-processador pula o código até o #endif

Implementação das funções

- A seguir, escrevemos o arquivo bibfunc.c, que contém a implementação das funções definidas anteriormente

```c
#include "bibfunc.h"

int fatorial(int v) {
    int res = 1, i;

    for (i = 1; i <= v; i++)
        res = res * i;

    return res;
}

int somatorio(int v) {
    int soma = 0, i;

    for (i = 1; i <= v; i++)
        soma += i;

    return soma;
}
```

- Note que este arquivo deve incluir o arquivo bibfunc.h

Programa principal

- Finalmente, o programa principal deve ser implementado em um arquivo separado, como por exemplo main.c:

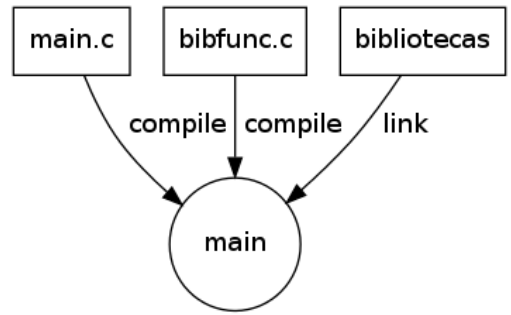
```c
#include <stdio.h>
#include "bibfunc.h"

int main()
{
    int v;

    printf("Digite um valor: ");
    scanf("%d", &v);
    printf("Fatorial: %d\n", fatorial(v));
    printf("Somatório: %d\n", somatorio(v));
    ...
}
```

- Observe que o arquivo bibfunc.h também deve ser incluído, pois contém a definição das funções

Compilação dos módulos

- Para compilar o programa completo, é preciso indicar todos os fontes na linha de comando:

```
gcc main.c bibfunc.c -o main
```

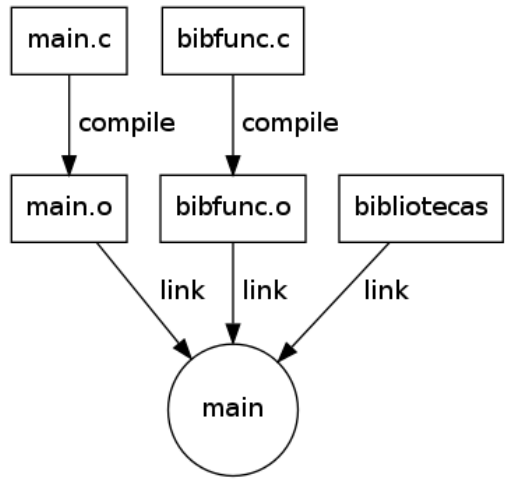


- Porém, essa estratégia tem algumas implicacões:
  - Todos os módulos são SEMPRE compilados novamente
  - Mas e se apenas mudar a implementação das funções?
  - E se apenas o programa principal mudar?
- Para um programa tão pequeno, isso não faz tanta diferença...
- Mas se o programa for composto por muitos módulos, compilá-los separadamente pode ter um custo considerável, além de não ser muito prático

Compilação modular

- Uma solução é compilar individualmente cada módulo:

```
gcc -c bibfunc.c
gcc -c main.c
```



- O comando *gcc -c...* apenas compila um módulo, gerando o código objeto (arquivo .o)
- Finalmente, geramos o programa executável, ligando os módulos e as bibliotecas do sistema:

```
gcc main.o bibfunc.o -o main
```

- Mas como decidir quais módulos precisam ser compilados?

Compilação modular: Makefiles

- Um *Makefile* é um roteiro de compilação:
  - Informa as dependências entre os arquivos
  - Indica os comandos necessários para a compilação
- Para utilizá-lo, basta digitar o comando *make* no terminal. O conteúdo do arquivo chamado *Makefile* é o seguinte:

```
all: main

main: main.o bibfunc.o
    gcc -o main main.o bibfunc.o

main.o: main.c
    gcc -c main.c

bibfunc.o: bibfunc.c
    gcc -c bibfunc.c
```

- O comando *gcc -c ...* apenas compila o módulo especificado, produzindo um arquivo .o, que é o chamado código-objeto (sem ligação com bibliotecas)

Melhorando o *Makefile*

- Os comandos de compilação não parecem meio repetitivos? Podemos melhorar o *Makefile*, criando regras genéricas:

```
CFLAGS = -Wall -O2

PROG = main
FONTES = main.c bibfunc.c
OBJETOS = $(FONTES:.c=.o)

$(PROG): $(OBJETOS)
    gcc $(CFLAGS) $(OBJETOS) -o $@

clean:
    -@ rm -f $(OBJETOS) $(PROG)
```

- **PROG = ...** define o nome do programa executável
- **FONTES = ...** define todos os módulos
- **OBJETOS = ...** define como transformar um código-fonte (.c) em um objeto (.o)
- **clean** permite apagar os arquivos .o e executável gerados (comando *make clean*)

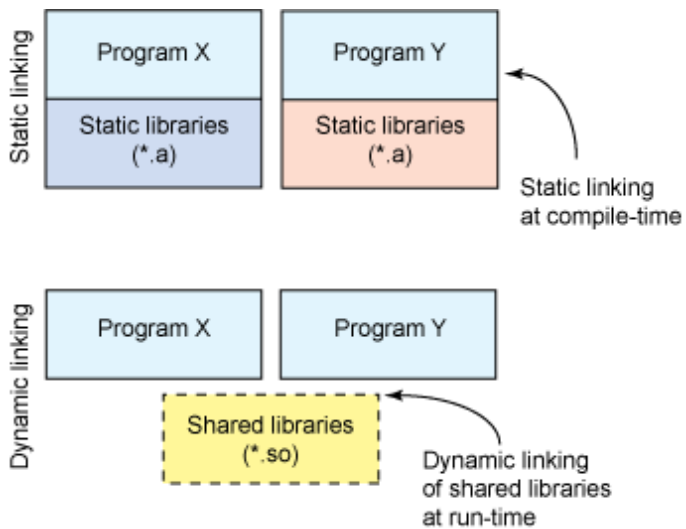
Alternativas ao *make*

- Já existem alguns sistemas alternativos[1] ao *make*:
  1. *CMake (Cross-Platform Make)* - este sistema tem o objetivo de gerar Makefiles adaptados para a plataforma de compilação
  2. *Scons* - escrito em Python, oferece uma sintaxe mais flexível, é multiplataforma e tem diversos recursos interessantes
  3. *QMake* - outro sistema semelhante, normalmente distribuído com o toolkit Qt, também multiplataforma
  4. *Boost.Build* - específico para C++, também multiplataforma e com uma sintaxe simples
  5. *GNU Build System (autotools)* - sistema mais utilizado atualmente para instalação de software no Linux/Unix (com software adicional instalado, pode ser utilizado também no Windows)

[1] Projetos criados em IDEs como *Visual Studio, Code::Blocks, Qt Creator, XCode, NetBeans,* etc... simplesmente automatizam essa tarefa.

Ligação estática x dinâmica

- Como já vimos, as bibliotecas são incluídas na etapa de ligação (linking), que é normalmente feita de forma automática pelo compilador
- Há duas formas de ligação:
  - *Estática*: os módulos são incorporados diretamente no código-objeto, isto é, o programa executável resultante conterá todo o código utilizado de cada módulo
  - *Dinâmica*: os módulos são referenciados no código-objeto, mas não incluídos no programa executável



Ligação estática

- A *ligação estática* é o que ocorre normalmente ao compilarmos um programa com vários módulos (exemplo: várias funções, separadas em arquivos diferentes)
- Cada código-fonte é compilado e produz um arquivo com código objeto equivalente (.c para .o, no caso do Linux)
- É possível também ligar bibliotecas de forma estática
  - Uma biblioteca nada mais é do que uma coleção de módulos, contendo funcionalidades relacionadas (ex: biblioteca de funções matemáticas, biblioteca gráfica, etc)
  - No Linux, bibliotecas estáticas são arquivos com o sufixo .a (de *archive*) e são criadas com o aplicativo *ar*
- As bibliotecas estáticas normalmente ficam nos diretórios */usr/lib* ou */usr/local/lib*. Por exemplo, verifique o conteúdo de */usr/lib*:

```
ls /usr/lib/*.a
```

Criando uma biblioteca estática

- Para criar uma biblioteca estática, utiliza-se a seguinte linha de comando:

  ```
  ar rcs libminhalib.a modulo1.o modulo2.o modulo3.o ...
  ```

  Onde *libminhalib.a* é o nome do arquivo resultante e *modulo*.o* são os diversos módulos a ser inseridos na biblioteca
- Uma vez criada a biblioteca, basta especificá-la[2] na linha de comando do compilador[3]:

  ```
  gcc -o programa programa.c -L. -lminhalib
  ```

- Considerando que *libminhalib.a* e *programa.c* estejam no mesmo diretório, esse comando irá utilizar a biblioteca durante a fase de ligação

[2] A biblioteca chama-se *libminhalib.a* mas é referenciada como *minhalib* na ligação

[3] O programa precisa ter acesso aos headers (.h) utilizados pela biblioteca

Ligação dinâmica

- Atualmente, a forma mais comum do uso de bibliotecas é através de *ligação dinâmica*
- Isso significa que o código executável (programa) não irá mais conter todo os módulos utilizados de cada biblioteca, mas apenas uma referência para eles
- No momento da execução, o sistema operacional identifica essas referências e carrega as bibliotecas necessárias, automaticamente
  - Isso torna o código menor, e permite que a implementação interna das bibliotecas sofra alterações, sem que seja necessário recompilar todos os programas que dependem dela
  - Isso também tem um efeito colateral benéfico: o sistema consegue manter uma cópia de cada biblioteca utilizada na memória, e compartilha ela com todos os programas que a utilizam

Criando uma biblioteca dinâmica

- Para criar uma biblioteca dinâmica, utiliza-se o próprio *gcc*

```
gcc -fPIC -shared -o libminhalib.so modulo1.o modulo2.o
    modulo3.o ...
```

- A opção *-fPIC* significa que o compilador deve gerar *position independent code*, isto é, código sem endereçamento absoluto, que pode ser carregado em qualquer ponto da memória
- Já a opção *-shared* indica a geração de uma biblioteca dinâmica
- Uma vez criada, a biblioteca dinâmica é utilizada da mesma forma que a estática

Utilizando uma biblioteca dinâmica

- A última etapa é adicionar uma regra para criar um executável que use a biblioteca dinâmica

```
gcc main.c -o main -L. -lminhalib
```

- Para utilizar a biblioteca dinâmica, será necessário adicioná-la em um diretório que contém outras bibliotecas ou adicionar o caminho onde está a biblioteca na variável de ambiente *LD_LIBRARY_PATH*

```
export LD_LIBRARY_PATH='pwd':$LD_LIBRARY_PATH
```

- Agora será possível executar a aplicação normalmente

Referências

- Este material foi construído com base nos slides de aula do prof. Marcelo Cohen