Notas de aula – AED 1 – bibliotecas e makefile
Prof. Jefferson T. Oliva

Nas aulas anteriores vimos conceitos de tipo abstrato de dados (TAD). Também, desenvolvemos TADs para estrutura de dados elementares, como lista, pilha e fila.

Na aula de hoje aprenderemos a construir bibliotecas, que podem conter vários TADs.

“Biblioteca é uma coleção de subprogramas utilizados no desenvolvimento de software.”

Em outras palavras, bibliotecas foram feitas para serem reutilizadas em outros projetos.

Desde que vocês começaram a programar na linguagem C, você pelo menos utiliza uma biblioteca padrão dessa linguagem, a ANSI C, que contém vários cabeçalhos (headers ou arquivos .h) as suas respectivas implementações (arquivos .c). Na biblioteca ANSI C, vários desses arquivos .h vocês devem ter utilizados, como stdio, stdlib, string, etc.

Assim, para podermos criar as nossas bibliotecas, precisaremos de arquivos .h e .c. Também, poderemos precisar de outras bibliotecas. Como utilizamos o compilador gcc, a extensão do arquivo que representa uma biblioteca é .a, caso o seu sistema operacional for Unix ou MacOS. Para o Windows, a extensão da biblioteca é .lib.

É importante ressaltar que as bibliotecas .a e .lib são bibliotecas estáticas, que são unidas ao programa em tempo de compilação.

As bibliotecas são bastante úteis em projetos, pois não precisamos “reinventar a roda”, ou seja, focar na solução de problemas.

Biblioteca

Biblioteca é um conjunto de funções ou interfaces empacotadas em um arquivo.

Para usá-las, basta chamarmos os seus respectivos arquivos .h através do comando #include, seja os que já estão localizados em diretórios pré-especificados pelo compilador (chamando através do comando <nome_arq.h>) ou no mesmo diretório do programa (chamando através do comando “nome_arq.h”). Para esse último caso, pode ser necessário informar o caminho completo do arquivo .h.

Para compilarmos e gerarmos bibliotecas na linguagem C, utilizamos o compilador GCC, que contém uma série de ferramentas para a realização de tarefas relacionadas à compilação, que inclui a geração de bibliotecas.

Por falar em compilação, na linguagem C, essa tarefa é realizada em etapas: pré-processamento, compilação, montagem e ligação. Caso não ocorra erros durante o processo (sintaxe, semântica, etc), um arquivo executável é gerado. No Windows, por exemplo, é gerado um arquivo .exe no final do processo de compilação.

Pré-processamento

Nessa etapa são tratadas as diretrizes (também denominadas diretivas), que são os comandos que não são compilados, mas são executados pelo compilador antes da continuação do processo de compilação. Essas diretrizes são iniciadas pelo caractere # (sharp ou cerquilha).

Durante o pré-processamento, as diretrizes modificam o programa fonte para deixá-lo pronto para ser compilado.

Você já deve ter utilizado algumas das diretivas da linguagem C, como #include e #define. Na aula de hoje veremos também outras diretrizes.

Inicie um arquivo .c, sem utilizar alguma diretiva, com seguinte fragmento de código.

```
int main(){
   prinf(“Oi! Eu sou o Gokku!\n”);
   return 0;
}
```

Em seguida, aplique o pré-processamento através do cpp conforme apresentado no exemplo abaixo.

cpp teste.c

Após, adicione a diretiva #include <stdio.h> no arquivo .c e execute o comando “*cpp teste.c*” novamente.

Como você pode observar, quando o cabeçalho stdio.h é “chamado” no arquivo .c, durante o pré-processamento todas as funções e diretivas relacionadas ao arquivo .h são verificadas durante o pré-processamento

Algumas diretivas de compilação utilizadas pelo pré-processador
- #include é utilizado para para chamar um arquivo específico
→ #include <nome_arquivo.h>: essa forma de uso significa um arquivo de cabeçalho é fornecido pelo compilador, que será incluído a partir de um diretório padrão do sistema
→ #include “nome_arquivo.h”: o arquivo foi feito pelo programador

O uso da diretiva #include facilita a leitura do código C ao abstrair detalhes de definições para uma outra etapa.

Outra diretiva que vocês devem ter utilizado é o #define, o qual usamos até o momento para definirmos constantes. Essa diretiva pode utilizada para determinarmos um símbolo (ou um macro) para ser utilizado durante o pré-processamento e a compilação

→ #define nome_simbolo
→ #define nome_constante valor
→ #define nome_macro(parâmetro) expressão_substituta

Outras diretivas utilizadas
- #undef: faz que uma macro seja esquecida, ou seja, a partir do ponto em que o compilador leu esse comando, passa a não conhecer mais o respectivo macro
→ #undef nome_macro

- #ifdef: operação condicional similar ao comando if

- #ifndef: operação condicional similar ao comando if, mas para o código em que o símbolo não foi definido
#ifndef SIMBOLO
#define SIMBOLO
#endif

- Outras diretivas: #if, #else e #elif (eslse if)

Exemplo de código com diretivas:

```
#include <stdio.h>
#define max(A, B) ((A > B) ? (A) : (B))

#ifdef MENSAGEM
	void mostrar(){
		printf("Inseto!\n");
	}
#else
	void mostrar(){
		printf("Kakaroto!\n");
	}
#endif

int main(){
	printf("Oi! Eu sou o Goku!\n!");
	mostrar();

	printf("\n%d\n", max(1,5));

	return 0;
}
```

Compilação

Nessa fase é realizada a análise sintática e semântica do código.

Pode ser realizado através do comando gcc no terminal.

Na compilação, as instruções C são transformadas em instruções assembly (tipicamente arquivos .s). Para que os arquivos assembly sejam acessíveis, deve ser utilizado o comando abaixo:

gcc -S algum_arquivo.c

Montagem

Também realizada através do comando gcc, transforma os arquivos com instruções assembly em um arquivo compilado .o, conhecido como arquivo "objeto", que são os arquivos arquivos binários.

Para a montagem, o comando gcc pode ser aplicado tanto em arquivos .c (os arquivos assembly são excluídos) quanto nos .s. Exemplos:

gcc -c algum_arquivo.s

gcc -c algum_arquivo.c

Ligação

Os arquivos .o são agrupados para gerar um arquivo executável e é nessa etapa que as bibliotecas são utilizadas.

Experimente utilizar o seguinte comando em um arquivo com main: *gcc nome_arquivo.c -o nome_executavel* ou *gcc nome_arquivo.o -o nome_executavel*

Biblioteca

No menu.h, por exemplo, você irá ver que que tem o seguinte trecho:

```
#ifndef _MENU_
#define _MENU_
…
#endif
```

Isso que foi feito é conhecido como guarda de cabeçalho, que serve para informar ao compilador que o seu cabeçalho foi incluído. O guarda de cabeçalho serve para proteção contra declarações duplicadas. Por isso, a recomendação é que você nomeie o cabeçalho adequadamente.

Compilação de arquivos .c e geração de arquivos .o:
gcc -c lista.c
gcc -c pilha.c
gcc -c fila.c
gcc -c menu.c
gcc -c principal.c

// Junção de todos os arquivos objetos em um executável
gcc -o saida menu.o principal.o lista.o fila.o pilha.o

// Criando uma biblioteca com o nome biblioteca.a. Caso estiver usando Windows, em vez de utilizar a extensão .a, use .lib
ar rs biblioteca.a lista.o pilha.o fila.o

// mostra quais arquivos objeto existem dentro de uma biblioteca
ar -t biblioteca.a

// compila o programa principal, junta com a biblioteca e gera o executável
gcc principal.c biblioteca.a menu.o -o saida

// A linha acima pode acarretar em erro porque as bibliotecas são procuradas em um diretório padrão
// Use: gcc principal.c -L./ -biblioteca menu.o -o saida

Makefile

O makefile é um arquivo texto com instruções de como um conjunto de arquivos fonte deve ser compilado.

O arquivo makefile é lido por um programa denominado make, que deve ser digitado na linha de comando. O programa make já vem instalado na maioria (senão em todos) dos sistemas operacionais.

Caso o arquivo makefile exista no diretório dos arquivos fonte, basta digitar make e pressionar a tecla enter.

Não é necessário digitar nome de arquivo, apenas o comando make, que irá procurar um arquivo texto com nome makefile e lê-lo.

Um makefile é utilizado para compilação, ligação e montagem.

No makefile também pode ser incluídas instruções para limpeza de arquivos temporários, execução de comandos, entre outros.

Vantagens
- Evita a compilação de arquivos desnecessários
- Automatiza tarefas, como limpeza de arquivos temporários
- Por mais que seja mais aplicado para compilação de arquivos, também pode ser utilizado como uma linguagem geral de script

No makefile contém regras que são definidas de acordo com a seguinte sintaxe:

alvo : pré-requisitos
<TAB>receita

→ Alvo é o nome da ação que será executada ou o nome do arquivo que deve ser produzido.
→ Pré-requisitos são os arquivos utilizados como entrada do nome_da_ação
→ Receita é a ação realizada pelo comando make
→ A receita pode ter mais de um comando
→ Também, os comandos podem ser atribuídos a uma variável, por exemplo: COMPILADOR = gcc
→ Para utilizar a variável que contém o nome do comando, basta utilizar $(NOME_VAR)

Após a definir e salvar o makefile, basta executar o comando make no diretório em que o arquivo encontra-se salvo.

Exemplo de makefile:

```
COMPILADOR=gcc
APAGA=rm -f
saida: principal.o menu.o lista.o pilha.o fila.o
	$(COMPILADOR) -o saida principal.o menu.o lista.o pilha.o fila.o

principal.o: principal.c menu.h lista.h pilha.h fila.h
	$(COMPILADOR) -c principal.c

menu.o: menu.h menu.c
	$(COMPILADOR) -c menu.c

lista.o: lista.h lista.c
	$(COMPILADOR) -c lista.c

pilha.o: pilha.h pilha.c
	$(COMPILADOR) -c pilha.c

fila.o: fila.h fila.c
	$(COMPILADOR) -c fila.c

clean:
	$(APAGA) saida *.o
```

Referências

de la Rocha, F. R. Bibliotecas. Slides. Engenharia de Computação. Dainf/UTFPR/Pato Branco, 2011.

Geeks for geeks. Gcc command in Linux with examples. https://www.geeksforgeeks.org/gcc-command-in-linux-with-examples/.

Ponti, M. P. Uma breve introdução à criação de bibliotecas e makefiles em C/C++. http://wiki.icmc.usp.br/images/0/0a/ApostilaMakefiles2011.pdf.

Wikibooks. Programar em C/Pré-processador. https://pt.wikibooks.org/wiki/Programar_em_C/Pr%C3%A9-processador.

Wikibooks. Programar em C/Makefiles. https://pt.wikibooks.org/wiki/Programar_em_C/Makefiles.