Aula de Laboratório 04

1 Compilação e arquivo Makefile[1]

Como vimos em aula, um arquivo de código fonte em linguagem C (.c) é compilado com o programa **gcc** (GCC - *GNU Compile Collection*). O GCC inclui *front ends* para as linguagens C, C++, Objective-C, Fortran, Ada, Go, e D, assim como as bibliotecas para essas linguagens [22]. Na primeira aula de laboratório, vimos que a compilação de um arquivo teste.c é realizado com o seguinte comando no terminal:

```
$ gcc -o executavel teste.c
```

Quando trabalhamos com mais de um arquivo de código fonte, temos que gerar os arquivos objetos de cada arquivo fonte e, em seguida, ligar os arquivos objetos para obtermos o arquivo binário (executável). A Figura 1 mostra uma simplificação do processo de compilação.

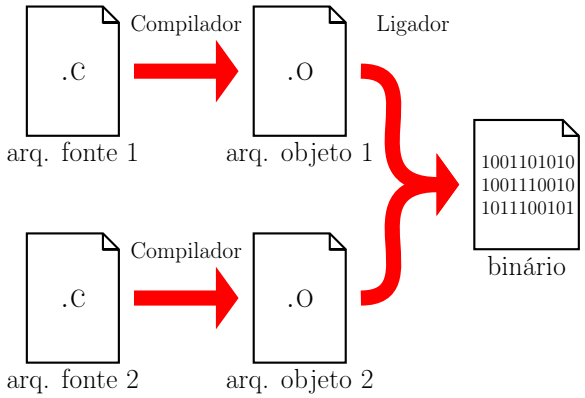

Figura 1: Representação simplificada da compilação de um programa. Fonte: https://www.embarcados.com.br/introducao-ao-makefile/

O programa **gcc** pode receber *flags* para controlar o pré-processador do C. Tais *flags* devem ser utilizadas pré-processamento de cada arquivo fonte antes da etapa final da compilação (ou seja, na compilação do arquivo de extensão .c para gerar o arquivo de extensão .o). Vimos em aula quatro *flags*: `-o`, `-Wall`, `-Werror` e `-Wextra`. Para mais informações sobre as *flags*, acessem a página do GCC.

Para ilustrar o processo de compilação de um programa composto por diferentes arquivos, vamos utilizar como exemplo os arquivos `aluno.h`, `aluno.c` e `main.c`. Os códigos são apresentados a seguir.

[1] As informações desta seção foram retiradas do site https://www.embarcados.com.br/introducao-ao-makefile/

Código do arquivo aluno.h

```
1  #ifndef __ALUNO_H__
2  #define __ALUNO_H__
3
4  /* Inclus o de bibliotecas necess rias para o pacote */
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8
9  /* Defini  o de estruturas e declara  o de vari veis locais */
10 typedef struct aluno {
11     int id;
12     char nome[100];
13     char email[200];
14     double media;
15 }Aluno;
16
17 /* Cabe alhos das fun  es com suas respectivas descri  es */
18
19 /*
20 Fun  o: criaAluno
21 Descri  o: Aloca dinamicamente uma estrutura aluno e inicializa os
       campos com os dados passados como par metro.
22 Entrada:
23     int id: identifica  o  nica  do aluno
24     char *nome: string com o nome do aluno. Tamanho m ximo  de
         100 caracteres.
25     char *email: string com o e-mail do aluno. Tamanho m ximo  de
          200 caracteres.
26     double media: nota do aluno no semestre.
27 Sa da:
28     ponteiro para a estrutura Aluno alocada din micamente
29 */
30 Aluno *criaAluno (int id, char *nome, char *email, double media);
31
32 /*
33 Fun  o: destroiAluno
34 Descri  o: libera a mem ria alocada para a vari vel passada como
       par metro
35 Entrada:
36     a: ponteiro para uma estrutura Aluno
37 Sa da: void
38 */
39 void destroiAluno (Aluno *a);
40
41 #endif
```

Código do arquivo aluno.c

```c
#include "aluno.h"

Aluno *criaAluno (int id, char *nome, char *email, double media){
    Aluno *novo = (Aluno *) malloc (sizeof(Aluno));
    if (!novo) {
        printf("Erro de alocacao.\n");
        exit(1);
    }
    novo->id = id;
    strcpy(novo->nome, nome);
    strcpy(novo->email, email);
    novo->media = media;

    return novo;
}

void destroiAluno (Aluno *a){
    free(a);
}
```

Código do arquivo main.c

```c
#include "aluno.h"

int main() {
    int identidade;
    char nome[100], email[200];
    double media;
    Aluno *a1 = NULL;
    printf("Digite o id do aluno: ");
    scanf("%d", &identidade);
    printf("Digite o nome do aluno: ");
    getc(stdin);
    scanf("%[^\n]", nome);
    printf("Digite o e-mail do aluno: ");
    getc(stdin);
    scanf("%[^\n]", email);
    printf("Digite a media do aluno: ");
    scanf("%lf", &media);
    a1 = criaAluno(identidade, nome, email, media);
    printf("id: %d\nnome: %s\ne-mail: %s\nmedia: %lf\n", a1->id, a1
       ->nome, a1->email, a1->media);
    destroiAluno(a1);
    return 0;
}
```

Para compilar o arquivo `aluno.c` devemos executar o seguinte comando no terminal:

```
gcc -c aluno.c -Wall -Werror -Wextra
```

Para compilar o arquivo `main.c` devemos executar o seguinte comando no terminal:

```
gcc -c main.c -Wall -Werror -Wextra
```

Para ligar os arquivos objetos gerados devemos executar o seguinte comando no terminal:

```
gcc -o teste aluno.o main.o
```

1.1 Makefile

O arquivo makefile consiste, principalmente, de regras definidas da seguinte forma:

```
alvo : pre_requisitos
<TAB>comando
```

- O `alvo` é o nome da ação que deseja-se executar ou o arquivo que deseja-se produzir.
- `pre_requisitos` é a lista de arquivos necessários para se executar o `comando`.
- `comando` é a instrução que a ser executada no terminal.
- É **obrigatório** usar o caractere <TAB> antes de qualquer comando no arquivo makefile.

A seguir temos o arquivo makefile referente à compilação dos arquivos vistos (aluno.h, aluno.c e main.c).

```
1
2  all: teste
3
4  teste: aluno.o main.o
5          gcc -o teste aluno.o main.o
6
7  aluno.o: aluno.c aluno.h
8          gcc -c aluno.c -Wall -Werror -Wextra
9
10 main.o: main.c aluno.h
11         gcc -c main.c -Wall -Werror -Wextra
12
13 clean:
14         rm -rf *.o teste
```

No terminal, digite: **"`make all`"**, o utilitário make vai executar o alvo `all` que se encontra na linha 2 do makefile. Este tem como pré-requisito o arquivo binário `teste`. O alvo teste, tem como pré-requisitos os arquivos aluno.o e main.o. Desta forma, precisamos dos respectivos alvos (implementados nas linhas 7 e 10). O alvo `aluno.o` precisa dos arquivos `aluno.c` e `aluno.h` para poder executar o comando associado a ele. Com a existência de tais arquivos, o comando da linha 8 é executado. Da mesma forma, o alvo `main.o` necessita dos arquivos `main.c` e `aluno.h` para executar o comando da linha 11.

Com isso, vimos um exemplo básico de arquivo makefile. Além da forma simples apresentada, é possível utilizar variáveis para facilitar as alterações e deixar o arquivo mais claro e reutilizável para outros projetos. Para mais detalhes, veja o site https://www.embarcados.com.br.

2 Pilha: Aplicação

Vimos em aula o TAD Pilha. Sua característica é: "o último elemento inserido é o primeiro elemento a ser retirado", ou seja, "*Last In, First Out* - LIFO". Uma das aplicações para a Pilha é a verificação do "casamento" de símbolos (delimitadores) de um programa. Por exemplo, nas sequências "(())" e "()()" os parênteses aparecem de maneira balanceada, enquanto nas sequências "(()))" e ")(" esse balanceamento não acontece. Nesse link, você encontra mais informações sobre parênteses balanceados. Considerando que os delimitadores aceitos em um programa são parênteses "(" e ")", colchetes "[" e "]" e chaves "{" "}", as declarações abaixo usam apropriadamente os delimitadores:

```
1    for(i=0; i<100; i++) {scanf("%d", &V[i]);}
2    a = V[10];
3    b = a*V[20];
4    c = (a+b)*(a-b)/V[2];
5    k = 2+(3-V[4])*(3-V[4]);
```

Já as declarações a seguir são exemplos do uso incorreto dos delimitadores.

```
1    while(V[a)>2]{a=c*(b+4));}
2    if (V[2+b)<4){return b;}
3    V[3] = k *(3-4*V[4);
```

Um delimitador em particular pode ser separado a partir de seu par por outros delimitadores, isto é, os delimitadores podem ser aninhados. Em consequência, um delimitador em particular está casado somente depois que todos os delimitadores que o seguem e que o precedem tenham sido casados.

O algoritmo de casamento de delimitador lê um caractere e o empilha em uma pilha se for um delimitador de abertura. Se um delimitador de fechamento é encontrado, ele é comparado a um delimitador que está no topo da pilha. Se eles se casam, o processamento continua. Caso contrário, o processamento para, assinalando um erro. O processamento termina com sucesso depois que todos os caracteres forem analisados e a pilha estiver vazia.

3 Tarefa

Questão 1. Implemente uma pilha para trabalhar com caracteres em arquivos pilha.c e pilha.h.

Questão 2. Escreva um programa que utiliza a pilha implementada na questão anterior para implementar o algoritmo descrito na Seção 2. A entrada do programa deve ser um arquivo de texto. O programa deve verificar se os delimitadores estão balanceados e imprimir na tela "Delimitadores balaceados" em caso positivo, e imprimir a mensagem "Delimitadores desbalanceados", em caso contrário.

4 Referências

Referências

[22] *GNU Compile Collection*. `https://gcc.gnu.org/`. Acessado em junho de 2022.