Modularização em C

Estruturas de Dados 2018/1
Prof. Diego Furtado Silva
Baseado no material do professor Gustavo E. A. P. A. Batista

Modularização em C

Programa em C pode ser dividido em vários arquivos

- Arquivos **fonte** (extensão **.c**)
  - Denominados *módulos*

Modularização em C

Cada módulo pode ser compilado separadamente

- Compilador gera **arquivos objeto** (não-executáveis)
  - Arquivos em linguagem de máquina ou **.o**

Modularização em C

Cada módulo pode ser compilado separadamente

- Compilador gera **arquivos objeto** (não-executáveis)
  - Arquivos em linguagem de máquina com extensão **.o**
- Arquivos objetos podem ser “juntados” em um **executável**
  - Para isso: ***linker*** ou ***link-editor***
  - Resultado: único arquivo em linguagem de máquina

Modularização em C

Módulos são muito úteis para a construção de **bibliotecas**

Ex:

- Módulo de funções matemáticas
- Módulo de manipulação de strings

Modularização em C

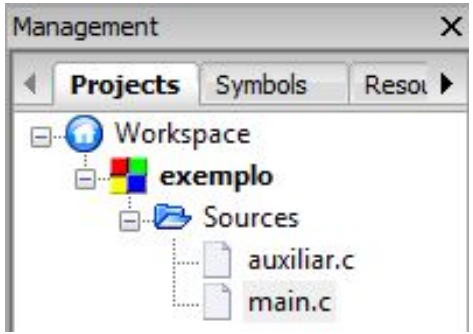
Primeiro, decidimos como o programa será dividido em módulos (arquivos):

- Cada TAD pode ser implementado em seu próprio módulo
- Outra forma é agrupar funções por propósito (bibliotecas, slide anterior)

Modularização em C

Em grandes projetos, uma possibilidade para organização do código é dividir a implementação em diversos arquivos

Modularização em C



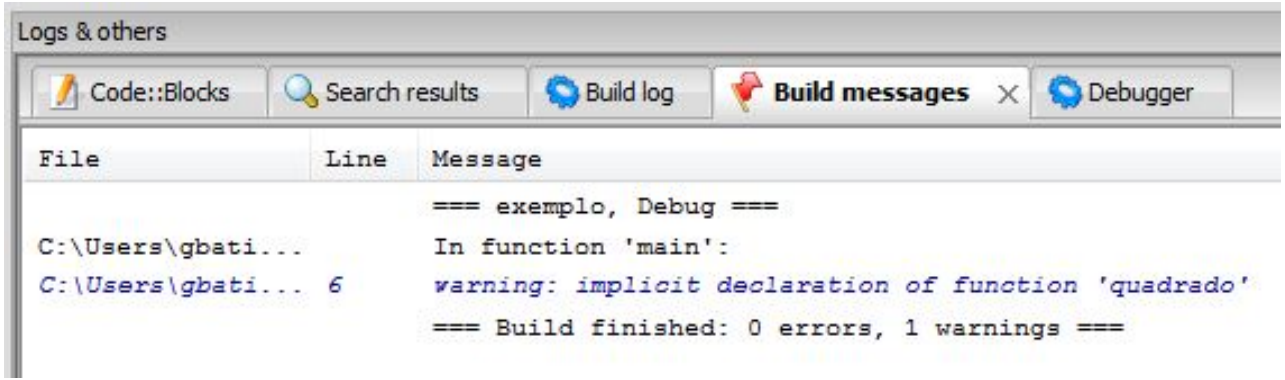

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    printf("%d\n", quadrado(2));
    return 0;
}
```

main.c

```c
int quadrado(int n)
{
    return n*n;
}
```

auxiliar.c



Modularização em C

Definir **protótipos** das funções

- Evita erros

```c
int quadrado(int n)
{
    return n*n;
}
```

auxiliar.c

```c
#include <stdio.h>
#include <stdlib.h>

int quadrado(int n);

int main()
{
    printf("%d\n", quadrado(2));
    return 0;
}
```

main.c

Modularização em C

Melhor ainda:

- Arquivos .h

```c
int quadrado(int n);

```

auxiliar.h

```c
#include "auxiliar.h"

int quadrado(int n)
{
    return n *  n;
}

```

auxiliar.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "auxiliar.h"

int main()
{
    printf("%d\n", quadrado(2));
    return 0;
}

```

main.c

Modularização em C

Melhor ainda:

- Arquivos **.h**
  - Múltiplas inclusões

```c
#ifndef AUXILIAR_H
#define AUXILIAR_H

int quadrado(int n);

#endif
```

auxiliar.h

```c
#include "auxiliar.h"

int quadrado(int n)
{
    return n *  n;
}
```

auxiliar.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "auxiliar.h"

int main()
{
    printf("%d\n", quadrado(2));
    return 0;
}
```

main.c

Modularização em C

Num segundo passo, decidimos quais partes serão públicas e quais serão privadas

- Partes públicas podem ser acessadas por outros módulos
- Partes privadas são acessíveis apenas dentro do módulo que as definiu

Público e privado

Em C, variáveis, tipos de dados definidos pelo usuário (*typedef*) e funções possuem escopo **global** (ou **público**)

- Se definidos em um arquivo, são visíveis em todos os demais
- Em vários compiladores exige-se que o protótipo da função seja definido no arquivo que a chama, e que a variável seja declarada como *extern*

Para simplificar, isso é normalmente feito no arquivo *header*

Extern

Uma variável global requer o uso de extern. Apesar das 2 declarações, trata-se da mesma variável

```c
#ifndef AUXILIAR_H
#define AUXILIAR_H

extern int a;

int quadrado(int n);

#endif
```

auxiliar.h

```c
#include "auxiliar.h"

int a = 100;

int quadrado(int n)
{
    return n *  n;
}
```

auxiliar.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "auxiliar.h"

int main()
{
    printf("%d %d\n", quadrado(2), a);
    return 0;
}
```

main.c

Público e privado

Por outro lado, algumas funções e variáveis devem escopo **interno** (ou **privado**)

- Somente são visíveis dentro do arquivo que as definiu
- Útil para funções e variáveis que fazem serviços internos e não são de interesse do restante do programa
- Em C requerem o uso do modificador *static* (que tem outro significado para variáveis locais)

Static (privado)

A função quadrado2 tem escopo interno ao arquivo auxiliar.c e não pode ser chamada de main.c

```c
#ifndef AUXILIAR_H
#define AUXILIAR_H

extern int a;

int quadrado(int n);

#endif
```

auxiliar.h

```c
#include "auxiliar.h"

static int quadrado2(int n)
{
    return n*n;
}

int quadrado(int n)
{
    return quadrado2(n);
}
```

auxiliar.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "auxiliar.h"

int main()
{
    printf("%d %d\n", quadrado(2), a);
    return 0;
}
```

main.c

Modularização em C - Compilando

A partir desses dois fontes (main.c e auxiliar.c), podemos gerar um executável compilando cada um separadamente e depois ligando-os

```
> gcc -c auxiliar.c
> gcc -c main.c
> gcc -o main auxiliar.o main.o
```

Modularização em C - Compilando

Ou podemos fazer tudo num passo só

```
> gcc -o main main.c auxiliar.c
```

Makefile

Forma prática de compilar um programa com vários arquivos

- Mais simples quando se tem vários arquivos fontes
- Mais rápido, pois permite recompilar somente os arquivos modificados

Makefile

O utilitário *make* é um programa que lê um arquivo de configuração (o Makefile), que define:

- Dependências entre arquivos
- Linha de comando para recompilar arquivos desatualizados

Makefile

Formato básico

```
regra (ou alvo): dependências
   comando
   comando

...
```

Dependência pode ser outra regra ou arquivo necessário

Makefile

Exemplo básico

```
programa: programa.c programa.h
	gcc -o programa programa.c
```

Ideal é usar o nome do programa de saída como nome da regra

- É assim que o make decide se vai recompilar o programa

Makefile

Exemplo menos básico

```
executa: programa
   ./programa

programa: programa.c programa.h
   gcc -o programa programa.c
```

Makefile

Exemplo monstrão

```
edit : main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
            gcc -o edit main.o kbd.o command.o display.o \
                       insert.o search.o files.o utils.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 buffer.h
            gcc -c display.c
insert.o : insert.c defs.h buffer.h
            gcc -c insert.c
search.o : search.c defs.h buffer.h
            gcc -c search.c
files.o : files.c defs.h buffer.h command.h
            gcc -c files.c
utils.o : utils.c defs.h
            gcc -c utils.c
clean :
            rm edit main.o kbd.o command.o display.o \
               insert.o search.o files.o utils.o
```

Makefile

Regra/alvo

Nome de um arquivo gerado por um programa

ou

nomes de ação

```
edit : main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
            gcc -o edit main.o kbd.o command.o display.o \
                       insert.o search.o files.o utils.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 buffer.h
            gcc -c display.c
insert.o : insert.c defs.h buffer.h
            gcc -c insert.c
search.o : search.c defs.h buffer.h
            gcc -c search.c
files.o : files.c defs.h buffer.h command.h
            gcc -c files.c
utils.o : utils.c defs.h
            gcc -c utils.c
clean :
            rm edit main.o kbd.o command.o display.o \
               insert.o search.o files.o utils.o
```

Makefile

Pré-requisito

Arquivos utilizados para gerar o alvo, que geralmente depende de vários pré-requisitos

```
edit : main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
            gcc -o edit main.o kbd.o command.o display.o \
                       insert.o search.o files.o utils.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 buffer.h
            gcc -c display.c
insert.o : insert.c defs.h buffer.h
            gcc -c insert.c
search.o : search.c defs.h buffer.h
            gcc -c search.c
files.o : files.c defs.h buffer.h command.h
            gcc -c files.c
utils.o : utils.c defs.h
            gcc -c utils.c
clean :
            rm edit main.o kbd.o command.o display.o \
               insert.o search.o files.o utils.o
```

Makefile

- Se um arquivo do alvo possui data e hora mais antigas do que qualquer um dos seus pré-requisitos, então a receita é executada para atualizar o alvo

Makefile

- Uma chamada a make (sem parâmetros) inicia uma verificação pelo primeiro alvo. Para ativar um alvo específico, utilize: make alvo, por exemplo:
  - *make clean*

- Clean é um alvo que não é um nome de arquivo, e não possui pré-requisitos. Por isso, make executa a receita diretamente

Makefile

Variáveis

- ajudam a simplificar o makefile reduzindo o risco de introduzir erros
- são definidas por meio do símbolo “=“
- são acessadas por meio de “$(variavel)”

Makefile

Variáveis

```
objects = main.o kbd.o command.o display.o \
              insert.o search.o files.o utils.o

edit : $(objects)
            gcc -o edit $(objects)
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 buffer.h
            gcc -c display.c
insert.o : insert.c defs.h buffer.h
            gcc -c insert.c
search.o : search.c defs.h buffer.h
            gcc -c search.c
files.o : files.c defs.h buffer.h command.h
            gcc -c files.c
utils.o : utils.c defs.h
            gcc -c utils.c
clean :
            rm edit $(objects)
```

Makefile

Variáveis e receitas reduzidas

- Para compilação, o *make* consegue deduzir as receitas pelo nome do alvo
- Ele utiliza o comando cc −c, por default

Makefile

```
objects = main.o kbd.o command.o display.o \
                insert.o search.o files.o utils.o

edit : $(objects)
              gcc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

clean :
              rm edit $(objects)
```

Makefile

```
CC=gcc
CFLAGS=-Wall
objects = main.o kbd.o command.o display.o \
                insert.o search.o files.o utils.o

edit : $(objects)
            $(cc) -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

clean :
            rm edit $(objects)
```

Makefile

- Tutorial simples, mas interessante

http://orion.lcg.ufrj.br/compgraf1/downloads/MakefileTut.pdf

TADs em C

Tipos abstratos de dados (TADs) podem ser implementados em C, utilizando módulos

- Cada TAD é implementado em um arquivo .c
- Um arquivo header (.h) deve ser feito com os protótipos das funções públicas e com as definições dos tipos de dados

TADs em C

Tipos abstratos de dados (TADs) podem ser implementados em C, utilizando módulos

- A implementação é “escondida” no arquivo .c. Funções e variáveis internas devem ter escopo interno (static)
- Pode-se prover apenas uma biblioteca com o .c compilado em código objeto + arquivo header

Exercício

- Reescrever o código do TAD Conjuntos (modularizado)

- Criar um Makefile para o TAD + aplicação