Módulo 7 - Projeto multi-folhas

Quando falamos em projetos multi-folha, significa que um mesmo programa está divido em vários arquivos `.c` **.** Este conhecimento serve tanto para organizar seu projeto quanto para criar suas próprias bibliotecas.

Em sua jornada como desenvolvedor de software, o ato de criar, modificar e utilizar bibliotecas se tonará rotina. Até agora fomos obrigados a utilizar a biblioteca que define as funções de entrada e saída pelo console em todas as aulas: `<stdio.h>`. Agora, aprenderemos a criar nossas próprias bibliotecas. As vantagens de se aprender a trabalhar com bibliotecas são várias:

1. **Reutilização de Código:** As bibliotecas fornecem uma coleção de funções e recursos prontos para uso. Ao aprender a usar essas bibliotecas, você pode reutilizar código já testado e otimizado, economizando tempo e esforço na implementação de funcionalidades comuns.
2. **Eficiência:** Bibliotecas escritas por especialistas tendem a ser altamente eficientes e otimizadas. Ao usar bibliotecas, você se beneficia da experiência de outros programadores, resultando em código mais rápido e eficiente.
3. **Desenvolvimento Mais Rápido:** O uso de bibliotecas pode acelerar o processo de desenvolvimento, permitindo que você se concentre nas partes específicas e únicas do seu projeto, em vez de reinventar a roda para tarefas comuns.
4. **Padrões e Boas Práticas:** Muitas bibliotecas são desenvolvidas seguindo padrões de codificação e boas práticas. Ao aprender a usar essas bibliotecas, você é exposto a abordagens de codificação recomendadas, melhorando sua própria prática de programação.
5. **Facilidade de Manutenção:** Usar bibliotecas padronizadas facilita a manutenção do código. Se houver uma atualização ou correção de bugs em uma biblioteca, você pode simplesmente atualizar a versão da biblioteca em seu projeto, em vez de ter que corrigir cada instância do código manualmente.

Mas antes de começar a construir bibliotecas, é necessário conhecer os pré-processadores da linguagem C:

Pré-processadores

Em linguagem C, um pré-processador é uma parte do compilador que lida com instruções de pré-processamento antes de o código ser compilado. Isso inclui diretivas de pré-processador que começam com `#`. Os pré-processadores são usados para realizar tarefas como inclusão de arquivos, definição de macros e condições de compilação condicional. Por serem processados antes mesmo da leitura do programa pelo compilador, é possível criar lógicas para fazer o compilador ignorar ou incluir trechos de códigos mediante condições, entre outros.

Aqui estão algumas das principais diretivas de pré-processador em C e seus usos comuns:

1. `#include`: Usado para incluir o conteúdo de um arquivo em seu código. Você informa qual arquivo quer referenciar entre `<>` ou `“”`, e o compilador incluirá tudo o que está dentro deste arquivo.

```
#include <stdio.h>
int main() {
    printf("Hello, World!\n");
    return 0;
}
```

2. `#define`: Define uma macro, que é um nome representando uma sequência de caracteres ou um valor. É muito parecido com uma variável, mas não é a mesma coisa. Uma **macro** não deve ter seu valor alterado durante o programa. Além disso, não é alocada na RAM, mas sim diretamente substituída no programa em tempo de compilação.

```
#define PI 3.14159
int main() {
    double raio = 5.0;
    double area = PI * raio * raio;
    return 0;
}
```

O `#define` também podem se comportar como funções, permitindo a definição de argumentos. A vantagem de utilizar este sistema é a economia de processamento e memória, já que o processador não precisa executar uma chamada de função. Veja um exemplo de uma definição de uma função soma:

```
#include <stdio.h>

#define SOMA(a, b) a + b

int main(void){
    printf("%d", SOMA(1, 5));

    return 0;
}
```

3. `#ifdef`, `#ifndef`, `#else`, `#endif`: Usado para compilação condicional. `#ifdef` verifica se um símbolo está definido, `#ifndef` verifica se um símbolo não está definido. É geralmente utilizado em conjunto com o `#define`. Importante: este condicional só funciona com macros, e não com variáveis comuns do programa.

```
#define DEBUG
#ifdef DEBUG
    // Código de depuração
    printf("Modo de depuração ativado\n");
#else
    // Código para lançamento
    printf("Modo de lançamento ativado\n");
#endif
```

Como criar uma biblioteca

O cabeçalho conterá os arquivos headers das funções (menos da função main) e todas as bibliotecas necessárias para a compilação do programa. Também é necessário encapsular esta folha com `#ifndef` para que não haja redefinição de nenhum

parâmetro. Também utilizamos a extensão `.h` no nome do arquivo para indicar que é um header, ou seja, que não será compilado.

- calculadora.h

```
#ifndef _CALCULADORA_H
#define _CALCULADORA_H

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int divide(int argv, char *argc[]);
int subtrai(int argv, char *argc[]);
int multiplica(int argv, char *argc[]);
int soma(int argv, char *argc[]);

#endif
```

O código fonte em si terá extensão `.c` e poderá ser dividido em quantas folhas forem necessário. Essas folhas deverão referenciar o arquivo cabeçalho de antes e deverão ser compiladas separadamente para, depois, serem linkadas. A folha principal, contendo a função `main`, geralmente é chamada de `main.c`.

- calculadora.c

```
#include "calculadora.h"

int soma(int argv, char *argc[]){
    int result = 0;
    for(int i = 2; i < argv; i++){
        result += atoi(argc[i]);
    }
    printf("%d", result);
}
```

```
//... continua
```

- main.c

```
#include "calculadora.h"

int main(int argv, char *argc[]){

    if(argv < 3){
        printf("Argumentos insuficientes");
        return 0;
    }

// ... continua
```

A partir de agora, sua biblioteca calculadora está pronta.

Como compilar um projeto multifolha

Para compilar essas diversas folhas de projeto que temos agora, é necessário compilar separadamente os códigos fonte e depois linka-los:

- `gcc -c main.c`
- `gcc -c calculadora.c`
- `gcc main.o calculadora.o -o calculadora`

Se para cada folha você precisa executar um comando de compilação, então um projeto de 10 folhas necessitará de 10 comandos de compilação. Felizmente, existe uma forma de automatizar todo este processo: makefile.

Makefile

O Makefile é um arquivo de configuração utilizado em sistemas operacionais Unix e Unix-like para automatizar o processo de compilação e construção de programas e projetos. Ele contém regras e instruções que indicam como compilar e vincular os diferentes componentes de um software.

Apesar de ser padrão Unix, podemos portá-lo para o windows com o auxílio do GCC. Iremos utilizar o programa **mingw32-make** já instalado no início deste curso. Ele deverá estar na pasta bin, dentro da pasta MinGW.

Precisamos apenas das regras para gerar os arquivos pré-compilados e depois uma ultima regra para linká-los:

```
all: main calculadora
    gcc main.o calculadora.o -o calculadora

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

calculadora:
    gcc -c calculadora.c -o calculadora.o
```

Estes comandos devem ser colocados em um arquivo chamado Makefile. Este arquivo não possui extensão e é reconhecido automaticamente pelo programa make.

A partir de agora, basta executar o mesmo programa **mingw32-make** para qualquer nova compilação que você queira realizar em seu software. Essas macros podem ser expandidas para qualquer tipo de projeto.