Aula 11 – Bibliotecas de função

1. Introdução

À medida que um programa cresce e fica mais complexo, o número de funções e de linhas de código em um único arquivo pode fazer com que sua compreensão e manutenção (adição de novas funções e correção de erros) fique mais difícil. Para facilitar, o compilador C permite a separação do código em arquivos diferentes, chamados "módulos".

2. Separação do código e inclusão

- Desde o tutorial de C vimos a diretiva de inclusão presente no código, para importar as funções da biblioteca padrão de operações de entrada e saída (`stdio.h`):

```
#include <stdio.h>

main() { /* ... */ }
```

- As bibliotecas padrão do C encontram-se em um diretório específico que o compilador conhece. Ao especificar o nome do arquivo entre < e >, o compilador procura naquele diretório;
- É possível, porém, especificar uma inclusão a partir do diretório atual, usando `""` ao invés de <>. Suponha o arquivo `prog.c`:

```
// Arquivo prog.c
#include "minhabib.c"

main() {
     minhaFunc();
}
```

- Para o código acima funcionar, a função `minhaFunc()` tem que ser definida no arquivo `minhabib.c`, que deve se encontrar na mesma pasta do arquivo `prog.c`:

```
// Arquivo minhabib.c
#include <stdio.h>

void minhaFunc() {
     printf("minhaFunc() foi chamada!\n");
}
```

- O compilador trata a inclusão de `minhabib.c` em `prog.c` como se todo o código estivesse num arquivo só (no caso, `prog.c`). Para compilar, portanto, basta chamar o compilador como de costume:

```
gcc -o prog prog.c
```

- Esse processo, no entanto, possui algumas falhas. Duas das quais:
    - Qualquer modificação nos arquivos envolvidos requer uma recompilação de todo o código. Para programas muito grandes, isso pode ser um processo demorado (de minutos a horas);
    - Não há flexibilidade para substituir uma implementação de um conjunto de funções por outra diferente.

3. Separação em módulos

- É mais recomendado criar um módulo, separando a biblioteca de funções em dois arquivos:
  - Arquivo de declarações ou arquivo cabeçalho (*header*) – extensão `.h`: declara as funções existentes naquela biblioteca, porém não provê sua implementação;
  - Arquivo de definições/implementações ou arquivo de código – extensão `.c`: define a implementação de todas as funções declaradas no arquivo cabeçalho;
  - Caso a biblioteca de funções utilize tipos de dados definidos pelo usuário, estes usualmente são definidos no arquivo cabeçalho.
- Tanto o arquivo da biblioteca de funções quanto o programa principal devem importar o arquivo cabeçalho, para garantir que a função implementada na biblioteca e chamada no programa seja a mesma (mesmo nome, parâmetros e tipo de retorno);
- Exemplo:

```
// Arquivo minhabib.h
void minhaFunc();
```

```
// Arquivo minhabib.c
#include <stdio.h>
#include "minhabib.h"

void minhaFunc() {
     printf("minhaFunc() foi chamada!\n");
}
```

```
// Arquivo prog.c
#include "minhabib.h"

main() {
     minhaFunc();
}
```

- Neste caso, como o programa principal importa somente o cabeçalho e não contém a definição (implementação) da função, ele não pode ser compilado sozinho. A compilação se dá em 2 passos:

```
gcc -c minhabib.c
gcc -o prog prog.c minhabib.o
```

- A compilação com a opção `-c` diz ao compilador que ali não existe um programa inteiro (e, portanto, ele não procura pela função `main()`), mas apenas uma biblioteca de funções. O compilador gera um arquivo `.o` (chamado de arquivo objeto) com o mesmo nome da biblioteca;
- Desta maneira, se precisarmos modificar somente o programa principal, a biblioteca de funções não terá que ser recompilada. E se tivermos 100 bibliotecas de função, se uma delas precisar ser modificada, somente ela e o programa principal precisarão ser recompilados;
- Se quisermos mudar a implementação de `minhabib.c` para `minhabib2.c`, se o arquivo cabeçalho permanecer o mesmo, basta trocar os comandos de compilação. O programa principal não precisará ser alterado;

4. Inclusão duplicada e condições de guarda

- À medida que o programa cresce e novas bibliotecas surgem, a rede de dependências entre as bibliotecas e o programa principal fica mais complexa;
- Pode acontecer a seguinte situação:

```
// Arquivo outrabib.h
typedef struct TMeuTipo {
     int inteiro;
     float real;
     char string[100];
} *MeuTipo;

void outraFunc(MeuTipo param);
```

```
// Arquivo outrabib.c
#include <stdio.h>
#include "outrabib.h"

void outraFunc(MeuTipo param) {
     printf("outraFuncao() foi chamada!\n");
}
```

```
// Arquivo minhabib.h
void minhaFunc(MeuTipo param);
```

```
// Arquivo minhabib.c
#include <stdio.h>
#include "minhabib.h"
#include "outrabib.h"

void minhaFunc(MeuTipo param) {
     printf("minhaFunc() foi chamada!\n");
}
```

```
// Arquivo prog.c
#include "minhabib.h"
#include "outrabib.h"

main() {
     MeuTipo var;
     minhaFunc(var);
     outraFunc(var);
}
```

- Note que o programa principal inclui o cabeçalho de `outrabib.h` duas vezes, uma diretamente e outra indiretamente (incluir `minhabib.h` e este inclui `outrabib.h`);
- Ao tentar compilar o programa principal, o seguinte erro é apresentado:

```
error: redefinition of 'struct TMeuTipo'
error: redefinition of typedef 'MeuTipo'
```

- Para evitar isso, usamos condições de guarda: diretivas do compilador são adicionadas ao arquivo cabeçalho para evitar que ele seja incluído duas vezes, desta forma:

```
// Arquivo outrabib.h
#ifndef __OUTRABIB_H
#define __OUTRABIB_H

typedef struct TMeuTipo {
     int inteiro;
     float real;
     char string[100];
} *MeuTipo;

void outraFunc(MeuTipo param);

#endif
```

- A diretiva `#ifndef` faz com que o compilador inclua o arquivo cabeçalho no ponto onde ele foi chamado somente se a constante `__OUTRABIB_H` não estiver definida;
- Caso ela realmente não esteja, a primeira coisa que o cabeçalho faz é defini-la, sem valor algum (porém, está definida);
- Da próxima vez que o cabeçalho for incluído, `#ifndef` retornará falso e o código do cabeçalho não será adicionado no ponto de inclusão.

5. Construção de programas com Makefile

- Também por conta do crescimento do número de bibliotecas de função, o processo de construção do programa (compilação dos arquivos) fica mais complicado;
- Uma ferramenta chamada `make` auxilia na automação do processo de construção:

```
all: prog

prog: prog.c minhabib.h outrabib.h minhabib.o outrabib.o
     gcc -o prog prog.c minhabib.o outrabib.o

minhabib.o: minhabib.c minhabib.h outrabib.h
     gcc -c minhabib.c

outrabib.o: outrabib.c outrabib.h
     gcc -c outrabib.c

clean:
     rm -rf *.o prog
```

- O arquivo acima chama-se `Makefile`. Funciona da seguinte maneira:
  - Os termos antes de "`:`" são os alvos. Quando digitamos "`make alvo`" ele procura o alvo na lista e executa os comandos;
  - Os termos após o "`:`" são as dependências. Ao tentar produzir um alvo, o `make` tenta primeiro satisfazer as dependências, que podem ser outros alvos;
  - Abaixo da declaração dos alvos e dependências estão os comandos. É essencial que cada linha de comando comece com uma tabulação;
  - O primeiro alvo deve sempre ser `all`, pois é o alvo padrão. Ao digitar o comando `make`, ele começará do alvo `all`. O alvo `clean` também é muito comum, ele apaga arquivos que são gerados pelo processo de construção;
  - Para os demais alvos, utilizamos os nomes dos arquivos produzidos a cada passo do processo de compilação;
  - Na lista de dependências devemos listar os arquivos que são incluídos com `#include` no código-fonte que está sendo compilado. Desta forma, se um destes arquivos sofrer alteração em relação à última compilação, o `make` sabe que deve repetir os comandos daquele alvo. Do contrário, ele não gasta tempo a toa;
  - Também devemos incluir os arquivos usados no comando de compilação (códigos fonte e arquivos objeto) pelo mesmo motivo acima (saber quando recompilar e quando não precisa).

6. Tipos abstratos de dados

- Utilizar como base slides do curso rápido de C, a partir do slide 194.