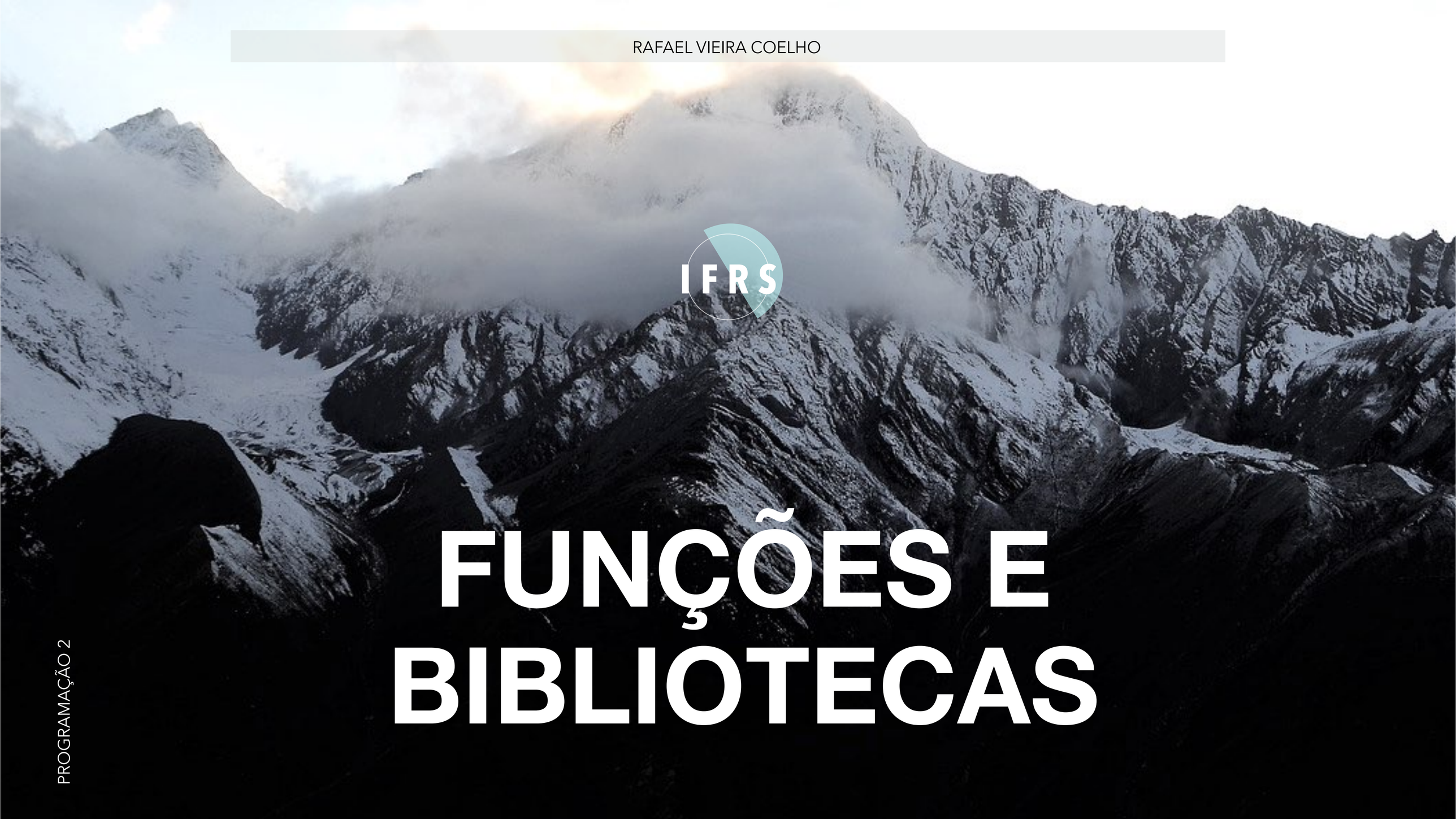
RAFAEL VIEIRA COELHO



FUNÇÕES E BIBLIOTECAS

PROGRAMAÇÃO 2

CAPÍTULO 19:ARGUMENTOS

https://books.goalkicker.com/CBook/

ARGUMENTOS DE LINHA DE COMANDO

O programa principal pode receber parâmetros quando executado por linha de comando (terminal).

A função main() recebe dois parâmetros:

argc (quantidade de argumentos - int)

***argv[]** (ponteiro para um vetor de parâmetros recebidos - char)

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

    for (int i = 1; i < argc; i++) {
        printf("Argument %d is: %s\n", i, argv[i]);

        errno = 0;
        char *p;
        long argument_numValue = strtol(argv[i], &p, 10);

        if (p == argv[i]) {
            fprintf(stderr, "Argument %d is not a number.\n", i);
        }
        else if ((argument_numValue == LONG_MIN || argument_numValue == LONG_MAX) && errno ==
ERANGE) {
            fprintf(stderr, "Argument %d is out of range.\n", i);
        }
        else {
            printf("Argument %d is a number, and the value is: %ld\n",
                   i, argument_numValue);
        }
    }
    return 0;
}
```

Obs: a função strtol() transforma string em inteiro longo

ARGUMENTOS DE LINHA DE COMANDO

O argumento **argv[0]** é o nome do programa que está sendo executado.

O segundo parâmetro do main() pode ser declarado como **char **argv** ou **char *argv[]**

Exemplo de como mostrar os argumentos recebidos:

```c
int main(int argc, char **argv)
{
    for (int i = 1; i < argc; i++)
    {
        printf("Argument %d: [%s]\n", i, argv[i]);
    }
}
```

ARGUMENTOS DO PROGRAMA PRINCIPAL

```c
#include <stdio.h>
#include <getopt.h>
#include <string.h>

/* print a description of all supported options */
void usage(FILE *fp, const char *path) {
    /* take only the last portion of the path */
    const char *basename = strrchr(path, '/');
    basename = basename ? basename + 1 : path;
    printf("usage: %s [OPTION]\n", basename);
    printf(" -h, --help\t\t"
           "Print this help and exit.\n");
    printf(" -m, --msg=STRING\t"
           "Output a particular message rather than 'Hello world'.\n");
}
```

ARGUMENTOS DO PROGRAMA PRINCIPAL

```c
/* parse command-line options and print message */
int main(int argc, char *argv[]) {
    /* for code brevity this example just uses fixed buffer sizes for strings */
    char message[256] = "Hello world";
    int help_flag = 0;
    int opt;
/* table of all supported options in their long form.
 * fields: name, has_arg, flag, val
 * `has_arg` specifies whether the associated long-form option can (or, in
 * some cases, must) have an argument. the valid values for `has_arg` are
 * `no_argument`, `optional_argument`, and `required_argument`.
 * if `flag` points to a variable, then the variable will be given a value
 * of `val` when the associated long-form option is present at the command
 * line.
 * if `flag` is NULL, then `val` is returned by `getopt_long` (see below)
 * when the associated long-form option is found amongst the command-line
 * arguments.
 */
    struct option longopts[] = {
            {"help", no_argument,       &help_flag, 1},
            {"msg",  required_argument, NULL,       'm'},
            {0}
    };
```

```c
/* infinite loop, to be broken when we are done parsing options */
    while (1) {
/* getopt_long supports GNU-style full-word "long" options in addition
     * to the single-character "short" options which are supported by
     * getopt.
     * the third argument is a collection of supported short-form options.
     * these do not necessarily have to correlate to the long-form options.
     * one colon after an option indicates that it has an argument, two
     * indicates that the argument is optional. order is unimportant.
     */
        opt = getopt_long(argc, argv, "hf::m:", longopts, 0);
        if (opt == -1) {
/* a return value of -1 indicates that there are no more options */ break;
        }
        switch (opt) {
            case 'h':
                /* the help_flag and value are specified in the longopts table,
                 * which means that when the --help option is specified (in its lo
                 * form), the help_flag variable will be automatically set.
                 * however, the parser for short-form options does not support the
                 * automatic setting of flags, so we still need this code to set t
                 * help_flag manually when the -h option is specified.
                 */
                help_flag = 1;
                break;
            case 'm':
/* since the argument for this option is required, getopt guarantees
             * that aptarg is non-null.
*/
                strncpy(message, optarg, sizeof(message));
                message[sizeof(message) - 1] = '\0';
                break;
            case '?':
                /* a return value of '?' indicates that an option was malformed.
                 * this could mean that an unrecognized option was given, or that
                 * option which requires an argument did not include an argument.
                 */
                usage(stderr, argv[0]);
                return 1;
            default:
                break;
        }
    }
```

ARGUMENTOS DO PROGRAMA PRINCIPAL

```
83      if (help_flag) {
84          usage(stdout, argv[0]);
85          return 0;
86      }
87      printf("%s\n", message);
88      return 0;
89  }
90
```

ARGUMENTOS DO PROGRAMA PRINCIPAL

Executando o programa:

```
[mac:Aula 7 - Funções e Bibliotecas coelho$ ./argumentos_linha_comando_e_arquivos
Hello world
[mac:Aula 7 - Funções e Bibliotecas coelho$ ./argumentos_linha_comando_e_arquivos -h
usage: argumentos_linha_comando_e_arquivos [OPTION]
  -h, --help              Print this help and exit.
  -m, --msg=STRING        Output a particular message rather than 'Hello world'.
[mac:Aula 7 - Funções e Bibliotecas coelho$ ./argumentos_linha_comando_e_arquivos -m Mensagem
Mensagem
[mac:Aula 7 - Funções e Bibliotecas coelho$ ./argumentos_linha_comando_e_arquivos -m Minha Mensagem
Minha
[mac:Aula 7 - Funções e Bibliotecas coelho$ ./argumentos_linha_comando_e_arquivos -m "Minha Mensagem"
Minha Mensagem
```

CAPÍTULO 25: FUNÇÕES

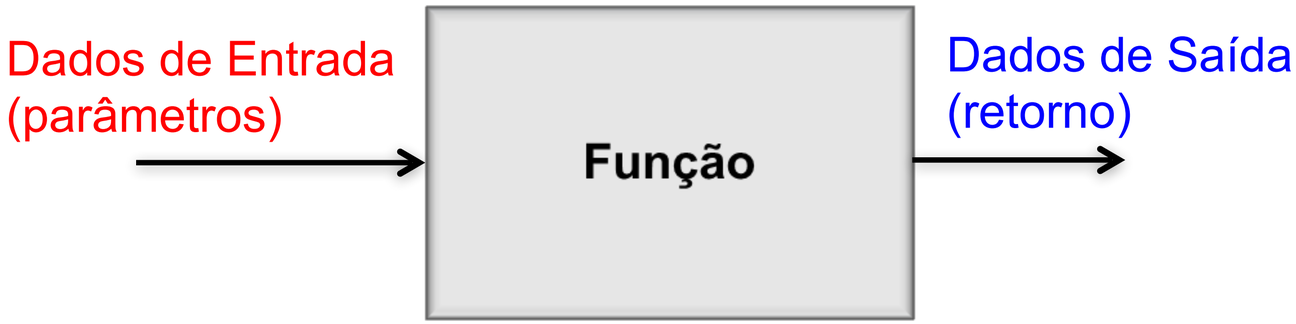
https://books.goalkicker.com/CBook/

Modularização de um Programa

- Um programa em C é um conjunto de funções, incluindo a função main();
- Funções são as estruturas que nos permitem organizar o código.
- Sem elas, somente os programas muito pequenos não ficariam complexos demais;
- O tipo de retorno padrão é int. Uma função que não retorna dados pode ser declarada como do tipo void.

A FORMA GERAL DE UMA FUNÇÃO

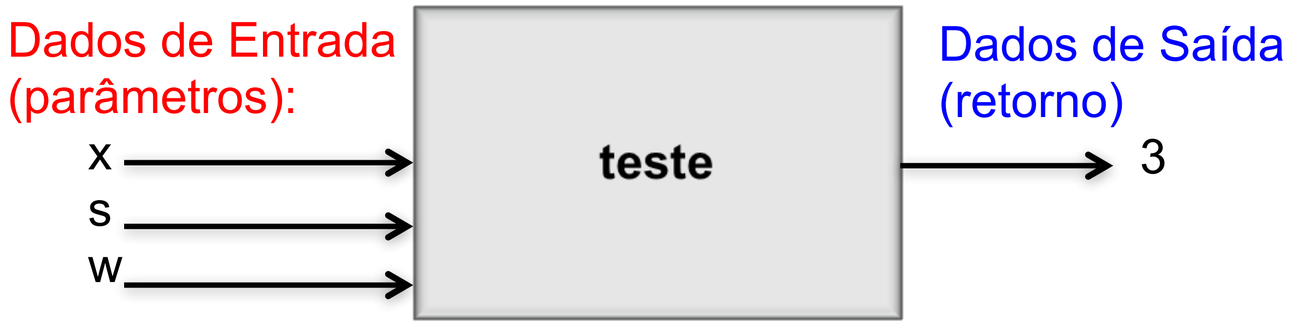
- tipo_saída **nome_função** (parâmetros_entrada){
  ....
- }



- - Os dados de entrada são dados para a função por quem a invocou
- - Os dados de saída é retornado pela função para quem a invocou

EXEMPLO DE UMA FUNÇÃO

```
int teste (int x, float s, char w){
        ....
        return 3;
}
```



- - Não existe um número limite para os dados de entrada. Inclusive pode não existir nenhum.

- - Os dados de saída podem ser tanto uma constante quanto uma variável

Modularização de um Programa

O comando return é usado para sair da função, retornando algum valor:

```
float quadrado(float x) {
        return x * x;
}
int imprimePositivo(float x) {
        if (x <= 0)
                return -1;
        return x;
}
int main() {
        printf("%f\n", imprimePositivo(quadrado(-4.7));
}
```

A função main

- A função main() é uma função como qualquer outra;
- Seu tipo de retorno é inteiro;
- O valor retornado por ela é disponibilizado ao sistema operacional (são chamados de exit codes);
- Há uma convenção na qual 0 (zero) significa término normal e qualquer número diferente de zero (de 1 a 256) significa algum erro;
- Alguns comandos especificam em seus manuais os diferentes códigos de erro existentes.

Declaração x Definição

- Declarar uma função é definir seu nome, tipo de retorno e argumentos (“protótipo” da função);
- Definir uma função é declará-la e ainda fornecer um corpo de código;
- Funções precisam ser declaradas antes de usadas, ou o compilador emitirá um erro em tempo de compilação;
- É útil para quando queremos separar nossas funções em arquivos (bibliotecas) que podem ser “importados”, assim como já fizemos com stdio.h e string.h.

Declaração x Definição – Exemplo

```
// Declaração (protótipo).
float quadrado(float x);
int main() {
        // Uso da função, só permitido porque ela já foi declarada.
        float f = quadrado(1.44);
}
// Definição necessária em tempo de ligação.
float quadrado(float x) {
        return x * x;
}
```

Escopo de Variáveis

- Trata-se de quais variáveis existem para o compilador naquele momento.

- O escopo das variáveis vai modificando conforme muda o contexto do programa (entrando e saindo de funções).

```
int global = 666;

int soma (int x, int y) {
      int s;

      s = x + y;
      return s;
}

int multiplica (int x, int y) {
      int m;

      m = x * y;
      return m;
}

void main() {
            int v1, v2, s, m;

            printf (“Informe o valor 1”); scanf(“%d”, &v1);
            printf (“Informe o valor 2”); scanf(“%d”, &v2);

            s = soma(v1, v2);
            m = multiplica(v1, v2);

            printf(“Soma = %d, Produto = %d”, s, m);
}
```

```c
int global = 666;

int soma (int x, int y) {
      int s;

      s = x + y;
      return s;
}

int multiplica (int x, int y) {
      int m;

      m = x * y;
      return m;
}

void main() {
            int v1, v2, s, m;

            printf (“Informe o valor 1”); scanf(“%d”, &v1);
            printf (“Informe o valor 2”); scanf(“%d”, &v2);

            s = soma(v1, v2);
            m = multiplica(v1, v2);

            printf(“Soma = %d, Produto = %d”, s, m);
}
```

Legenda:

Variáveis Locais

Variáveis Globais

Passagem de Parâmetro

- **Por valor:**

```
void imprimeValor(int x) {
        x++;
        printf("%d\n", x);
}
int main() {
        int x = 10;
        printf("%d\n", x);  // 10
        imprimeValor(x);   // 11
        printf("%d\n", x);  // 10
}
```

- **Por referência (com ponteiros):**

```
void imprimeValor(int *x) {
    (*x)++;
    printf("%d\n", *x);
}
int main() {
    int x = 10;
    printf("%d\n", x);    // 10
    imprimeValor(&x);  // 11
    printf("%d\n", x);    // 11
}
```

Vetores como Argumentos

- Quando se passa um vetor por parâmetro, ele é sempre passado por referência:

```
void imprimeValor(int x[]) {
        x[0]++;
}
void imprimeValor2(int *x) {
        x[0]++;
}
```

```
int main() {
        int y[2];

        y[0] = 10;
        y[1] = 20;
        printf("Antes %d\n", y[0]);
        imprimeValor(y);
        printf("Depois do imprimeValor():
%d\n", y[0]);
        imprimeValor2(y);
        printf("Depois do imprimeValor2():
%d\n", y[0]);
        return 0;
}
```

CAPÍTULO 44: BIBLIOTECAS

https://books.goalkicker.com/CBook/

Bibliotecas (Header Files)

- Bibliotecas são arquivos que contém somente declarações (protótipos) de funções;
- Possuem a extensão “.h”
- Podemos criar nossas próprias bibliotecas de função
- Para importar arquivos de cabeçalho definidos pelo programador, usamos " " ao invés de <>:

  #include "minhabiblioteca.h"

Criando bibliotecas

- funcoes.h:

```
float quadrado(float x) {
        return x * x;
}
```

- funcoes.c:

```
#include "funcoes.h"

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

Principais Bibliotecas

Arquivo de Cabeçalho	Finalidades
assert.h	Define a macro *assert()*
ctype.h	Manipulação de caracteres
errno.h	Apresentação de erros
float.h	Define valores em ponto flutuante dependentes da implementação
limits.h	Define valores em ponto flutuante dependentes da implementação
locale.h	Suporta localização
math.h	Diversas definições usadas pela biblioteca de matemática
setjmp.h	Suporta desvios não-locais
signal.h	Suporta manipulação de sinal
stdarg.h	Suporta listas de argumentos de comprimento variável
stddef.h	Define algumas constantes normalmente usadas
stdio.h	Suporta E/S com arquivos
stdlib.h	Declarações miscelâneas
string.h	Suporta funções de *strings*
time.h	Suporta as funções de horário do sistema

BIBLIOTECA FOO.H

foo.h

```c
#ifndef FOO_DOT_H    /* This is an "include guard" */
#define FOO_DOT_H    /* prevents the file from being included twice. */
                     /* Including a header file twice causes all kinds */
                     /* of interesting problems.*/


/**
 * This is a function declaration.
 * It tells the compiler that the function exists somewhere.
 */
void foo(int id, char *name);

#endif /* FOO_DOT_H */
```

IMPLEMENTAÇÃO FOO.C

foo.c

```c
#include "foo.h"    /* Always include the header file that declares something
                     * in the C file that defines it. This makes sure that the
                     * declaration and definition are always in-sync.  Put this
                     * header first in foo.c to ensure the header is self-contained.
                     */
#include <stdio.h>

/**
 * This is the function definition.
 * It is the actual body of the function which was declared elsewhere.
 */
void foo(int id, char *name)
{
    fprintf(stderr, "foo(%d, \"%s\");\n", id, name);
    /* This will print how foo was called to stderr - standard error.
     * e.g., foo(42, "Hi!") will print `foo(42, "Hi!")`
     */
}
```

PROGRAMA PRINCIPAL: MAIN.C

main.c

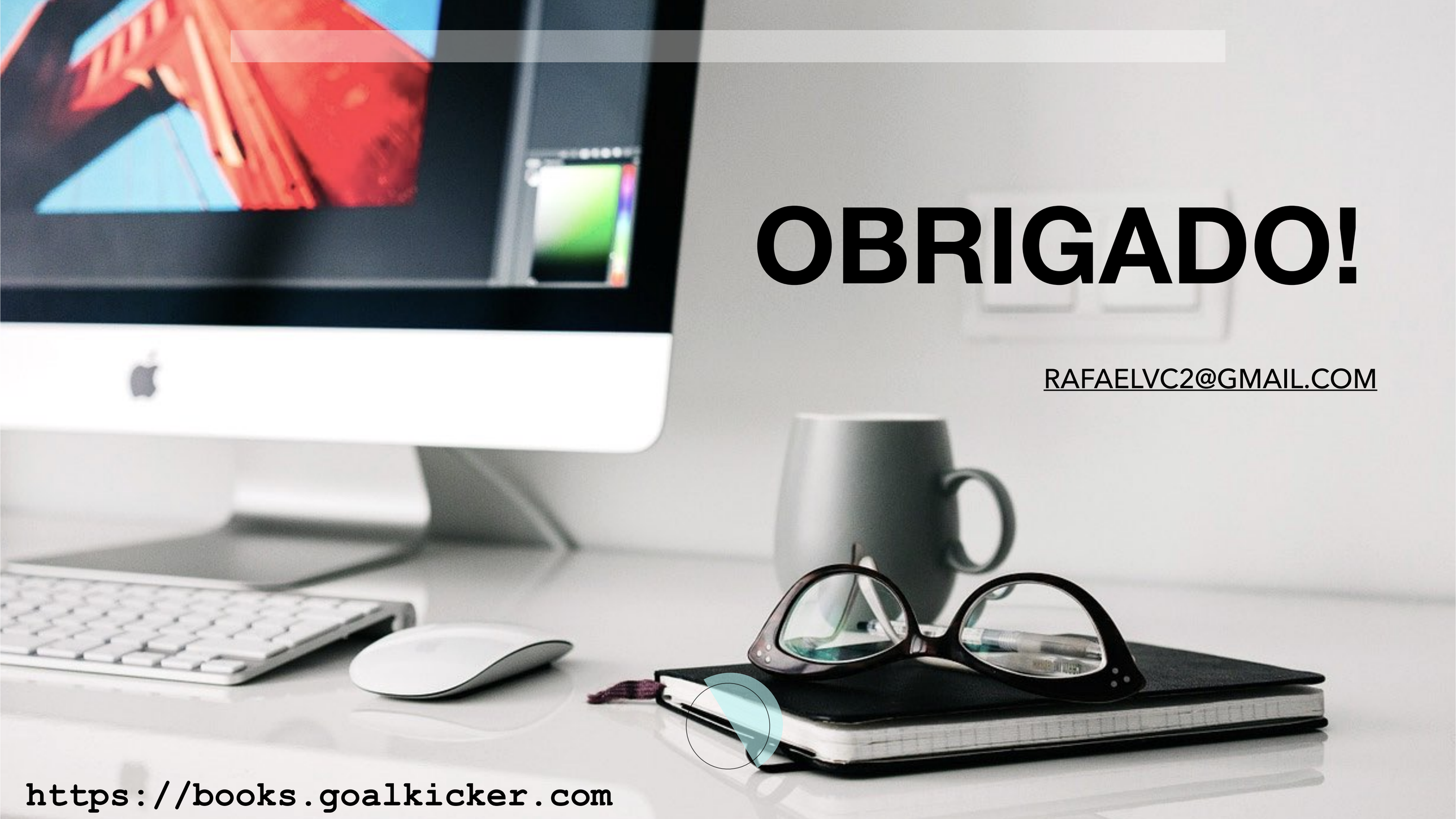
```c
#include "foo.h"

int main(void)
{
    foo(42, "bar");
    return 0;
}
```

Inicialmente, precisamos compilar ambos arquivos.

```
$ gcc -Wall -c foo.c
$ gcc -Wall -c main.c
```

Posteriormente, podemos fazer a linkagem.

```
$ gcc -o testprogram foo.o main.o
```



OBRIGADO!

RAFAELVC2@GMAIL.COM

https://books.goalkicker.com