


Introdução a Técnicas de Programação

Aula 14
Misc: I/O, .h,...





Leitura e escrita em arquivos



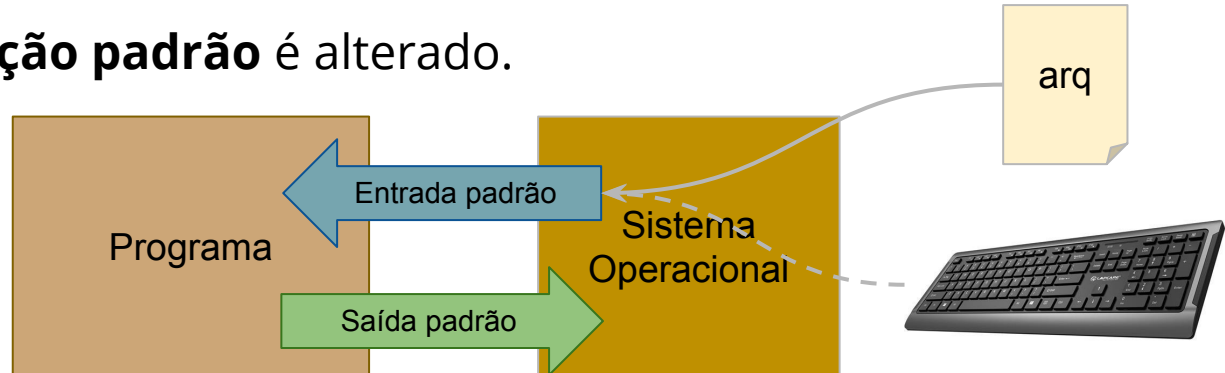
Canais de comunicação

Já vimos como ler e escrever de arquivos usando o redirecionamento da entrada e saída na linha de comando.

```
$ ./main < entrada.txt  
$ ./main > saida.txt  
$ ./main < entrada.txt > saida.txt
```

O que acontece nesses redirecionamentos?

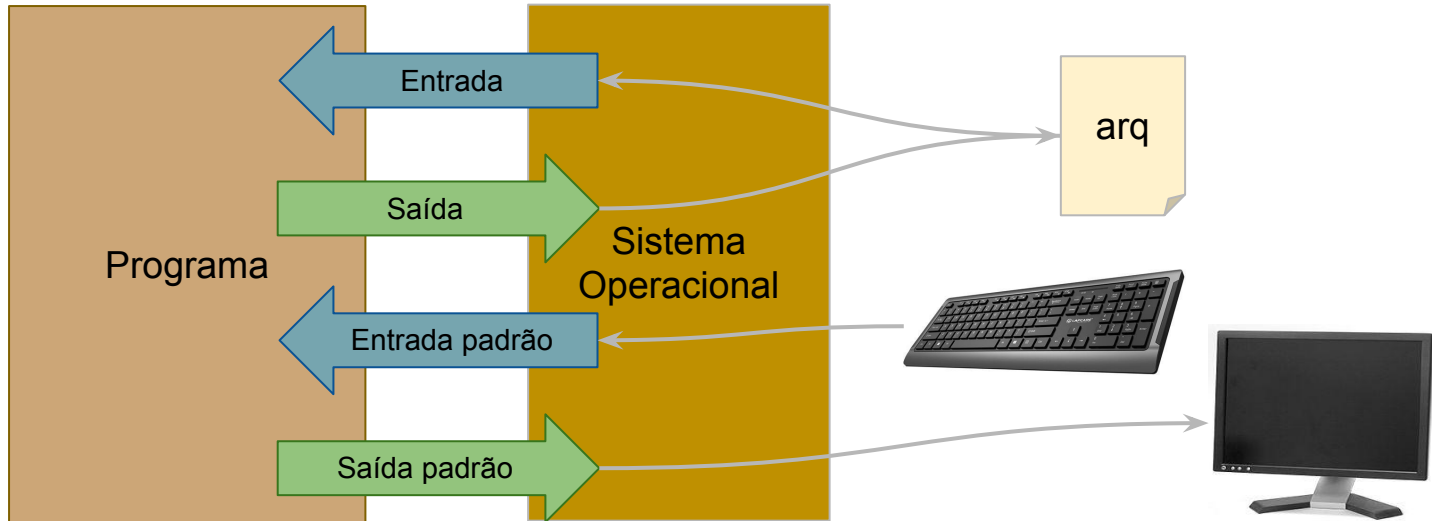
O **canal de comunicação padrão** é alterado.



Canais de comunicação

Podemos criar novos canais de comunicação (streams)
Associados a arquivos, podemos ler e escrever em arquivos

- Sem precisar alterar o canal de comunicação padrão!!!



Processo

Fluxo natural de uso dos canais de comunicação (stream):

1. Abertura

- É necessário indicar o modo de operação: leitura, escrita, texto, binário...

2. Leitura ou escrita de dados

- Similar à leitura e escrita na entrada e saída padrão (semelhante a `scanf()` e `printf()`)

3. Fechamento

- **É importante “fechar”** o canal quando não for mais necessário acessar os dados porque:
 - i. É um recurso (memória e processamento) que está sendo utilizado.
 - ii. Quando há uma escrita de dados, eles são guardados inicialmente no *buffer* (memória temporária). Ao fechar, o *buffer* é esvaziado e o arquivo atualizado. Se o programa terminar sem fechar, o arquivo pode ficar sem a última atualização.

Arquivos em C

Há um tipo (struct) já predefinido em `<stdio.h>` para manipular os canais de comunicação e associar a um arquivo: `FILE`.

Normalmente, declaramos um ponteiro para a estrutura `FILE` para ser a **referência do arquivo**, pois a função de abertura retorna um ponteiro.

Essa referência é usada nas funções de leitura, escrita e para fechar o canal.

```
#include <stdio.h>

void main() {
    FILE *file;
    file = fopen("entrada.txt", "r");
    ...
    // leitura dos dados
    ...
    fclose(file);
}
```

Modos de abertura

Modo	Operação
r (ou “rb” para binários)	Read. Leitura de arquivo
w (ou “wb” para binários)	Write. Escrita de arquivo. Cria se não existir, escreve a partir do início do arquivo.
a (ou “ab” para binários)	Append. Escrita de arquivo. Cria se não existir, escreve a partir do final do arquivo.
r+ (“rb+” ou “r+b” para binários)	Leitura e escrita. Escreve do começo, mas não trunca. Só abre se existir.
w+ (“wb+” ou “w+b” para binários)	Leitura e escrita. Escreve do começo, trunca o arquivo. (Igual a w)
a+ (“ab+” ou “a+b” para binários)	Leitura e escrita. Escreve do final do arquivo;

Erros de abertura

É importante testar se a abertura do canal deu certo

- Arquivos inexistentes
- Arquivos em que o usuário não tem permissão para ler/escrever

A função `fopen()` retorna...

- um ponteiro para o canal de comunicação alocado, se deu certo
- a constante `NULL`, se deu errado

```
FILE *file;  
file = fopen("entrada.txt", "r");  
if (file == NULL) {  
    printf("Erro na abertura!\n");  
}
```


Leitura e escrita

Similar à entrada padrão, porém com um parâmetro a mais: a referência para o arquivo:

- `scanf()` → `fscanf()`
- `printf()` → `fprintf()`
- `gets()` → `fgets()`
- ...

Funções mais usadas:

- `fscanf(arquivo, formatação, variáveis);`
- `fprintf(arquivo, formatação, variáveis);`

Ex. de escrita

Salva no arquivo
"respostas.txt" as
opções de resposta de
um aluno.

```
#include <stdio.h>

int main() {
    int i, alternativa;
    FILE *arquivo = fopen("respostas.txt", "w");
    if (arquivo == NULL) {
        printf("Erro na abertura do arquivo.\n");
        return 1;
    } else {
        printf("Digite as alternativas (opções de 1 a 4).\n");
        for (i = 0; i < 10; i++) {
            printf("Questão %d: ", i + 1);
            scanf("%d", &alternativa);
            fprintf(arquivo, "Resposta %d: %d\n", i + 1, alternativa);
        }
        fclose(arquivo);
    }
    return 0;
}
```

Ex. de leitura

Lê do arquivo "respostas.txt" as opções de resposta de um aluno e imprime o número de acertos de acordo com um gabarito.

```
#include <stdio.h>

int main() {
    int questao, resposta, acertos, i;
    int gabarito[10] = {3, 4, 2, 1, 3, 1, 1, 2, 4, 4};
    FILE *arquivo = fopen("respostas.txt", "r");
    if (arquivo == NULL) {
        printf("Erro na abertura do arquivo");
        return 1;
    } else {
        acertos = 0;
        for (i = 0; i < 10; i++) {
            fscanf(arquivo, "Resposta %d: %d\n", &questao, &resposta);
            if (gabarito[questao - 1] == resposta) acertos++;
        }
        printf("Acertos: %i\n", acertos);
        fclose(arquivo);
    }
    return 0;
}
```

Erros de leitura e escrita

As funções `fprintf()` e `fscanf()` retornar valores que podem ser testados.

`fscanf()`

- Se deu certo, retorna o número de itens lidos da lista de variáveis.
- Se o texto formatado não “casou”, retorna 0 (nada lido)
- Se houve erro durante a leitura, retorna um valor negativo
 - Pode retornar a constante EOF (= -1) se tentou ler mas já estava no fim do arquivo.

`fprintf()`

- Se deu certo, retorna o número de caracteres escritos.
- Se houve erro durante a leitura, retorna um valor negativo

Outras funções

Há várias outras funções para leitura e entrada de dados em arquivos. Pode-se salvar registros completos, salvando blocos inteiros da memória, salvar dados em binários (despreza a codificação ASCII), etc.

- `fwrite()` → escreve um array de elementos de um bloco de memória
- `fread()` → lê um array de elementos
- `ftell()` → consulta a posição de leitura/escrita do arquivo
- `fseek()` → reposiciona a leitura/escrita do arquivo
- `feof()` → verifica se chegou no final do arquivo
- ...

Organização de código

Organizar... pra quê?

“Hábito saudável” de qualquer profissão.
Economia cognitiva

Em programação:

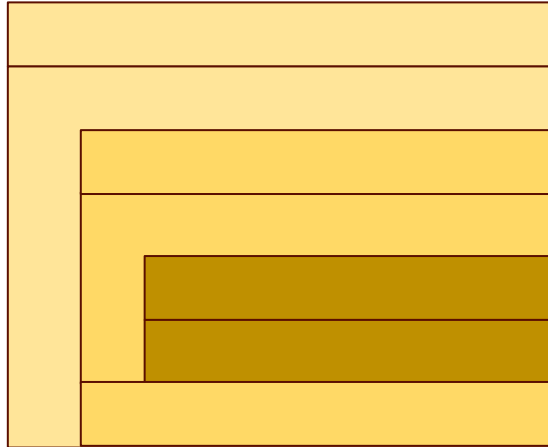
- Para facilitar a leitura e consequente manutenção do código (por você ou terceiros);
- Para dividir o trabalho (organização lógica de tarefas);
- Para não repetir a mesma tarefa e reutilizar algo já feito;
- ...



Organização visual

Identação

- Termo dado ao espaço no início dos parágrafos
- Em programação, é uma forma de organizar o código para ajudar a leitura
- É fácil identificar quais blocos de instruções são subconjuntos de quais outros blocos.



Organização visual

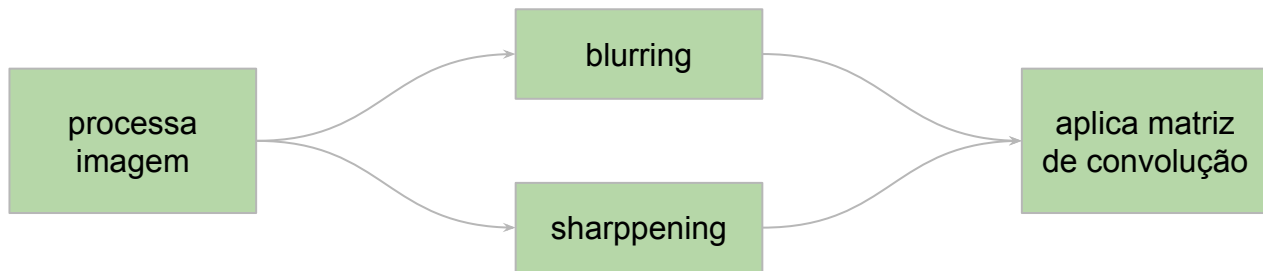
Padronização do estilo

- Útil quando se trabalha em equipe (+1 pessoas)
- Convenção mútua: sabe-se o que se espera quando lê o código de outros
- Exemplos
 - Constantes em maiúsculas
 - Nomes de variáveis e funções em minúsculas
 - Uso de `_` para separar nomes (ex: `num_pessoas`) ou iniciais maiúsculas (ex: `numPessoas`)
 - Número de espaços nas identações
 - Início da definição de blocos na mesma linha (ex: `if(...) {`) ou na linha seguinte
 - Funções pequenas (que dê para ler toda sem precisar “rolar” a tela do editor)
 - ...

Organização lógica

Modularização

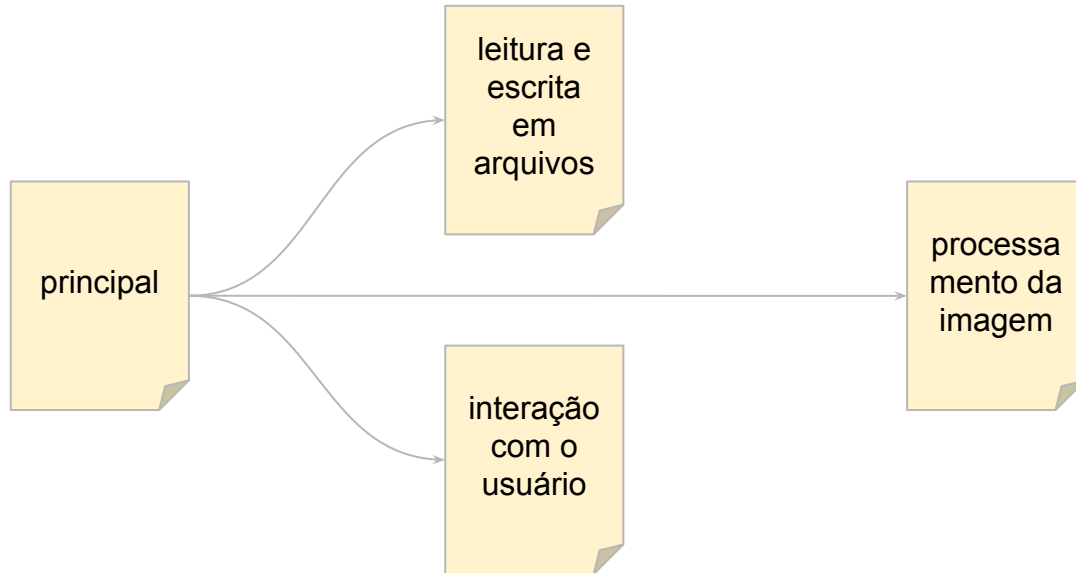
- Divisão do programa em partes lógicas (cada uma com sua função)
- Reuso de código



Organização lógica

Modularização com múltiplos arquivos

- Agrupamento de funcionalidades de um programa
- Facilita manutenção (projetos com +500 funções: “onde está função x???”)



Múltiplos arquivos em C

Criação de módulos

- permite que apenas as partes do código alteradas sejam compiladas
- permite a divisão de trabalho por diferentes membros de uma equipe

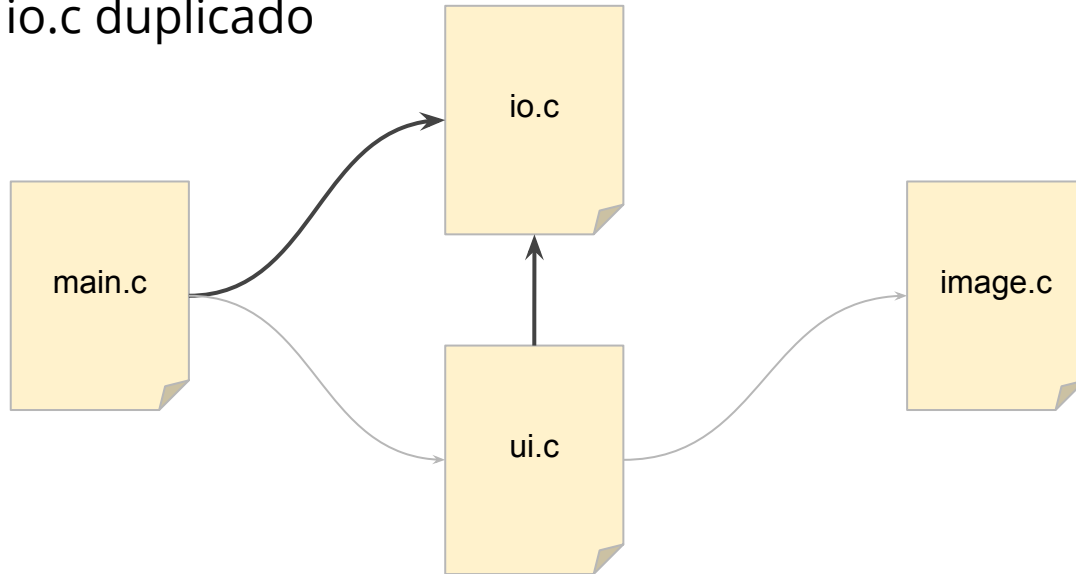
Criação de bibliotecas

- reuso de soluções pré-compiladas em diferentes projetos

Módulo em C

Cada grupo de funcionalidades (módulo) pode estar implementada em um .c

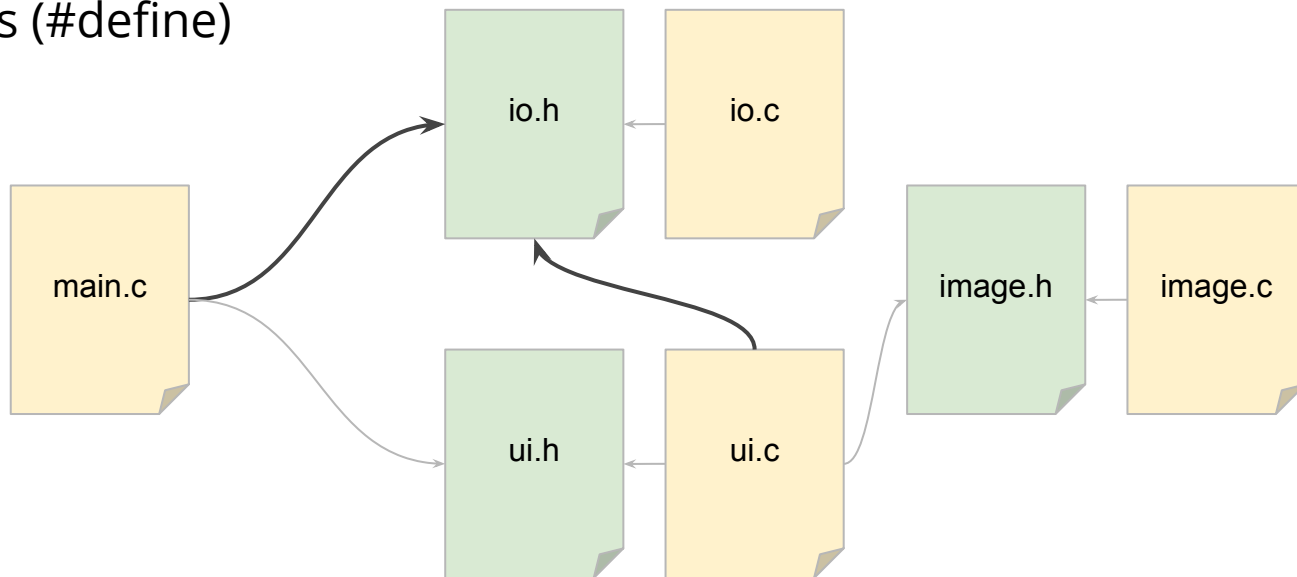
- Porém, não é uma boa prática incluir o .c através do #include, pois pode haver duplicação de código
- Ex: código io.c duplicado



Solução: arquivos cabeçalhos (.h)

Arquivos que possuem apenas definições (e não implementações):

- Assinatura de funções
- Definição de tipos (struct e enum)
- Constantes (#define)



Exemplo 1

main.c

```
#include "console.h"  
...
```

console.h

```
char* readString();  
char readChar();  
int readInt();  
float readFloat();  
void printChar(char c);  
void printInt(int i);  
void printFloat(float f, unsigned int n);  
void print(char *format, ...);
```

console.c

```
#include "console.h"  
  
char* readString() {  
    // implementação da função  
}  
  
char readChar() {  
    // implementação da função  
}  
  
int readInt() {  
    // implementação da função  
}  
...
```

Exemplo 2

main.c

```
#include <stdio.h>
#include "point.h"
void main() {
    Point p1 = { 5.3, 7.8 };
    Point p2 = { -8.2, 9.1 };
    printf("%.2f\n", distance(p1, p2));
}
```

point.h

```
typedef struct {
    float x, y;
} Point;

float distance(Point a, Point b);
```

point.c

```
#include <math.h>
#include "point.h"

float distance(Point a, Point b) {
    float dx = a.x - b.x;
    float dy = a.y - b.y;
    return sqrt(dx*dx + dy*dy);
}
```


Compilação

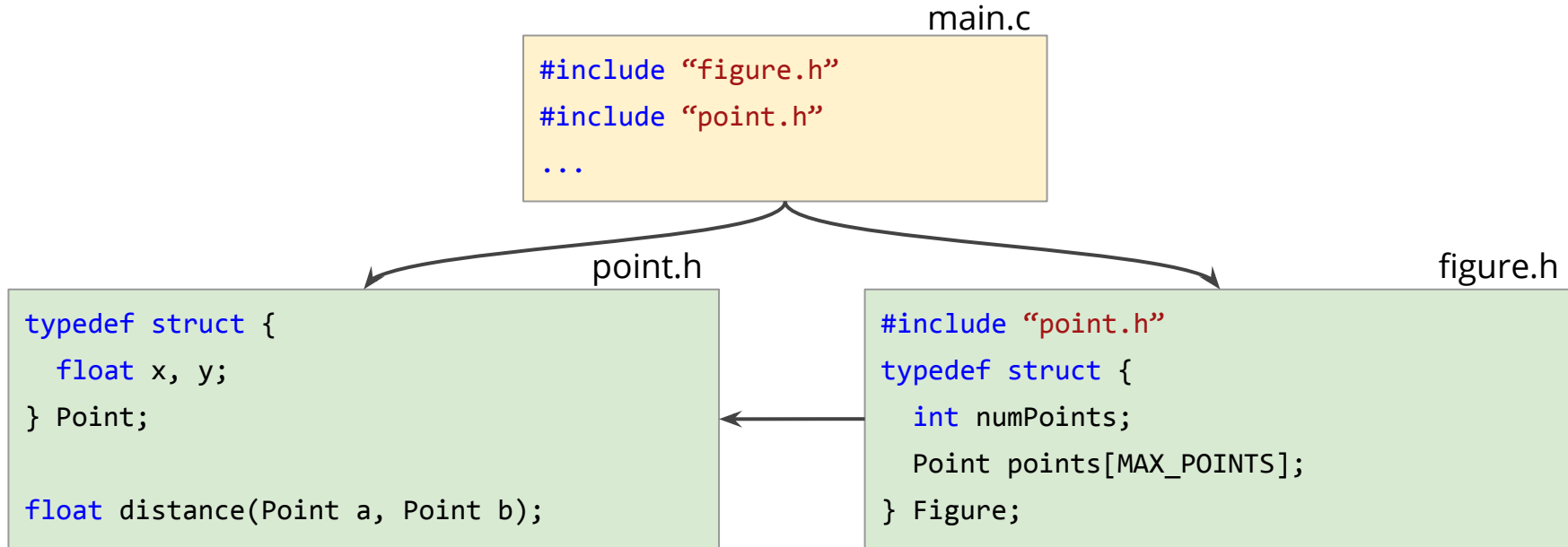
Não há ligação entre os .c, então como saber que eles fazem parte do mesmo programa????

⇒ No momento da compilação! Compila-se apenas os arquivos .c

```
$ gcc main.c modulo1.c modulo2.c -o executavel
```

Mais um problema...

E se um arquivo .h precisar de outro arquivo .h e um .c incluir os dois?
No exemplo, o tipo Point será definido duas vezes.



Solução: include guard

Uso de diretivas para o compilador não processar um arquivo.

- `#ifndef` → verificar se uma macro foi definida
- `#define` → define uma macro
- `#endif` → termina um bloco de código iniciado por `#ifndef` (ou `#ifdef`)

Na 1ª vez que for processar, `POINT_H` não foi definido, então `#ifndef POINT_H` é verdadeiro e o bloco é processado.

Na 2ª vez, `POINT_H` já foi definido, então `#ifndef` é falso e o bloco não é processado.

```
#ifndef POINT_H
#define POINT_H

typedef struct {
    float x, y;
} Point;

float distance(Point a, Point b);

#endif
```

Práticas

1. Escreva um programa para verificar se uma palavra se encontra em um arquivo. O usuário deve digitar a palavra e o nome do arquivo onde ela será procurada. Se a palavra se existir no texto, deve-se imprimir “sim”. Caso contrário, deve-se imprimir “não”.
2. Escreva um programa para saber em que parágrafo de um texto se encontra uma palavra dada pelo usuário. O usuário deve digitar a palavra e o nome do arquivo onde ela será procurada. Para cada parágrafo que contiver a palavra do usuário, o programa deve imprimir o parágrafo.
 - a. Cada parágrafo é uma linha do arquivo e não há especificação do número de linhas.
 - b. A função `getline()` lê uma linha inteira.
3. Escreva um programa para indicar as palavras de um texto que não existem no português. O usuário deve digitar o nome do arquivo que contém o texto e o programa deve ler de outro arquivo as palavras em português (<http://www.dimap.ufrn.br/~andre/itp/palavras.txt>). Toda palavra que não contém português deve ser impressa pelo programa.