Métodos de ordenação e busca em arranjos

com uma visão geral sobre pré-processador, Makefile e funções recursivas

Alexsandro Santos Soares
prof.asoares@gmail.com

Universidade Federal de Uberlândia
Faculdade de Computação

Introdução

- Uma das aplicações mais comuns na computação é a ordenação.
- Ordenação é o processo pelo qual os dados são arranjados de acordo com seus valores.
- Nesta aula veremos dois algoritmos de ordenação, seleção e inserção, aplicados a arranjos.
  - Já vimos esses algoritmos no início do semestre quando usamos a analogia das cartas de baralho.
- Também veremos algoritmos para encontrar um determinado elemento em um arranjo.

Método da seleção

Vamos relembrar o método da seleção:

Algoritmo 1: Ordena por Seleção

```
1  Coloque a pilha de cartas sobre uma mesa
2  enquanto existir carta na mesa faça
3  |   Com a mão direita pegue a menor carta na mesa
4  |   se há carta na mão esquerda então
5  |   |   Coloque a nova carta à direita da última carta na mão esquerda
6  |   senão
7  |   |   Coloque a nova carta na mão esquerda
8  |   fim se
9  fim enqto
10 retorna a pilha de cartas na mão esquerda
```

Método da seleção aplicado a arranjos

- Um arranjo modelará a *pilha de cartas*.
- Dividiremos o arranjo em duas partes:
  - Usando o lado esquerdo do arranjo, os primeiros elementos, modelamos a *mão esquerda*, a parte já ordenada.
  - O lado direito do arranjo, os demais elementos, fará a vez da *mão direita*, a parte não ordenada.
- Com essas mudanças o pseudocódigo paro o método da seleção para arranjos é mostrado no próximo slide.

Pseudocódigo para o método da seleção

Algoritmo 2: Ordena Arranjo por Seleção

1 Coloque os elementos a serem ordenados no arranjo
2 $i \leftarrow 0$
3 **enquanto** *não chegar ao final do arranjo* **faça**
4 | Selecione o menor elemento no lado direito do arranjo `/* de i+1 até o final */`
5 | Permute o menor elemento com o i-ésimo elemento do lado esquerdo
6 | $i \leftarrow i + 1$ `/* o lado esquerdo aumenta em 1 e o direito diminui em 1 */`
7 **fim enqto**
8 **retorna** o arranjo ordenado

Exemplo 1 (Ordena por seleção)

A seguir apresentamos o código em C para a ordenação de arranjos em ordem crescente usando o método da seleção.

```
21 void ordenaSelecao(int a[], int tam){
22   int imenor = 0; // índice do menor elemento
23   int temp = 0;  // temporário usado para realizar a permuta
24
25   for(int i = 0; i < tam; i++){
26     // Selecione o menor elemento no lado não ordenado
27     imenor = i;
28     for(int j = i + 1; j < tam; j++)
29       if (a[j] < a[imenor])
30          imenor = j;
31
32     // Permute o menor elemento com o primeiro elemento
33     // no lado não ordenado.
34     temp = a[i];
35     a[i] = a[imenor];
36     a[imenor] = temp;
37   } // for i
38
39   return;
40 } // ordenaSelecao
```

```
42 int main(void){
43   int i = 0;
44   int a[10] = { 10, 5, 1, 2, 6, 4, 3, 7, 9, 8};
45
46   printf("\nArranjo original:\n");
47   for(i = 0; i < 10; i++)
48     printf("%d ",a[i]);
49   printf("\n");
50
51   ordenaSelecao(a, 10);
52
53   printf("Arranjo ordenado:\n");
54   for(i = 0; i < 10; i++)
55     printf("%d ",a[i]);
56   printf("\n");
57
58   return 0;
59 } // main
```

Uso

```
> ./exemplo1.exe

Arranjo original:
10 5 1 2 6 4 3 7 9 8
Arranjo ordenado:
1 2 3 4 5 6 7 8 9 10
```

Método da inserção

Vamos relembrar também o método da inserção:

Algoritmo 3: Ordena por Inserção

```
1  Coloque a pilha de cartas sobre uma mesa
2  enquanto existir carta na mesa faça
3  |   Com a mão direita pegue a carta no topo
4  |   se há carta na mão esquerda então
5  |   |   Coloque a nova carta na posição correta da mão esquerda
6  |   senão
7  |   |   Coloque a nova carta na mão esquerda
8  |   fim se
9  fim enqto
10 retorna a pilha de cartas na mão esquerda
```

Pseudocódigo para o método da inserção

Usando a mesma analogia empregada no método da seleção, o pseudocódigo para o método da inserção aplicado a arranjos é:

Algoritmo 4: Ordena Arranjo por Inserção

```
1 Coloque os elementos a serem ordenados no arranjo.
2 i ← 1
3 enquanto existirem elementos desordenados faça
4 |   temp ← i-ésimo elemento do arranjo
5 |   Procure a posição correta para inserir temp no lado esquerdo movendo os
  |     elementos uma posição para a direita, se necessário.
6 |   Insira temp na posição correta do lado esquerdo.
7 |   i ← i + 1
8 fim enqto
9 retorna o arranjo ordenado
```

Exemplo 2 (Ordena por inserção)

A seguir apresentamos o código em C para a ordenação de arranjos em ordem crescente usando o método da inserção.

```
23 void ordenaInsercao(int a[], int tam){
24   int i = 0; // índice da posição correta
25   int temp = 0; // temporário usado para realizar a permuta
26
27   for(int j = 1; j < tam; j++){
28     // O lado esquerdo a[0..j-1] já está ordenado
29     temp = a[j];
30
31     // Procure a posição correta para inserir temp no lado esquerdo,
32     // movendo os elementos para a direita, se necessário.
33     i = j - 1;
34     while (i >= 0 && temp < a[i]){
35       a[i+1] = a[i]; // move o elemento uma posição para a direita
36       i = i - 1;
37     } // while
38
39     a[i+1] = temp; // insira temp na posição correta do lado esquerdo
40   } // for j
41
42   return;
43 }
```

```
45 int main(void){
46   int i = 0;
47   int a[10] = { 10, 5, 1, 2, 6, 4, 3, 7, 9, 8};
48
49   printf("\nArranjo original:\n");
50   for(i = 0; i < 10; i++)
51     printf("%d ",a[i]);
52   printf("\n");
53
54   ordenaInsercao(a, 10);
55
56   printf("Arranjo ordenado:\n");
57   for(i = 0; i < 10; i++)
58     printf("%d ",a[i]);
59   printf("\n");
60
61   return 0;
62 } // main
```

Uso

```
> ./exemplo2.exe

Arranjo original:
10 5 1 2 6 4 3 7 9 8
Arranjo ordenado:
1 2 3 4 5 6 7 8 9 10
```

Busca em arranjos

- Uma outra operação comum em computação é a busca.
- Uma busca é o processo usado para encontrar a localização de um elemento alvo no meio de um arranjo de elementos.
- A busca consiste em encontrar o índice do primeiro elemento no arranjo que contenha um dado elemento.
- Existem duas formas básicas de buscas em arranjos:
  - a sequencial, que pode ser utilizada para encontrar um item em qualquer arranjo.
  - a binária, que requer que o arranjo esteja ordenado.

Busca sequencial

- A busca sequencial é utilizada quanto o arranjo não estiver ordenado.
- Essa técnica geralmente é usada para arranjos pequenos ou que não sejam pesquisados com frequência.
  - Nos outros casos, devemos primeiro ordenar o arranjo e depois usar a busca binária que será discutida em breve.
- Na busca sequencial começamos a procurar o alvo a partir do início do arranjo e continuamos a busca até encontrarmos o alvo ou concluirmos que ele não está no arranjo.
  - Ou seja, temos duas possibilidades: encontramos o alvo no arranjo ou chegamos ao final do arranjo.
- Projetaremos a função de busca sequencial de tal forma que:
  - se o alvo estiver no arranjo ela devolve `true` e insere o índice do elemento encontrado em um endereço passado como parâmetro.
  - Se o alvo não estiver no arranjo ela devolve `false`.

Exemplo 3 (Busca sequencial)

A seguir apresentamos o código em C da função que realiza uma busca sequencial em um arranjo tentando localizar um dado elemento.

```
28 bool buscaSequencial(int a[], int tam, int alvo, int *pPos){
29   int i = 0; // índice do arranjo
30   bool encontrado = false;
31
32   i = 0;
33   while (i < tam && alvo != a[i])
34     i++;
35
36   *pPos = i;
37   encontrado = (alvo == a[i]);
38
39   return encontrado;
40 } // buscaSequencial
```

```
43 int main(void){
44   int i = 0;
45   int pos = 0;
46   int num = 0;
47   int a[10] = { 10, 5, 1, 2, 6, 4, 3, 7, 9, 8};
48
49   printf("\nArranjo original:\n");
50   for(i = 0; i < 10; i++)
51     printf("%d ",a[i]);
52   printf("\n");
53
54   printf("\nDigite o número a ser localizado: ");
55   scanf("%d", &num);
56
57   if (buscaSequencial(a, 10, num, &pos))
58     printf("%d foi encontrado no índice %d\n", num, pos);
59   else
60     printf("%d não foi encontrado no arranjo\n", num);
61
62   return 0;
63 } // main
```

Uso

```
> ./exemplo3.exe

Arranjo original:
10 5 1 2 6 4 3 7 9 8

Digite o número a ser localizado: 2
2 foi encontrado no índice 3

> ./exemplo3.exe

Arranjo original:
10 5 1 2 6 4 3 7 9 8

Digite o número a ser localizado: 0
0 não foi encontrado no arranjo
```

Busca binária

- O algoritmo de busca sequencial é muito lento para grandes arranjos.
  - Se tivéssemos um arranjo com 1 milhão de elementos precisariámos fazer 1 milhão de comparações no pior caso.
- Se o arranjo estiver ordenado, podemos usar uma algoritmo mais eficiente chamado de busca binária.
- A busca binária começa testando o elemento do meio do arranjo.
  - Isso determina se o alvo está na primeira ou na segunda metade do arranjo.
  - Se o alvo estiver na primeira metade não precisamos verificar a segunda metade e vice-versa.
  - De qualquer modo, deixaremos de testar metade do arranjo.
- O processo é repetido na metade de interesse até encontrar o alvo ou concluir que ele não está no arranjo.

Exemplo 4 (Busca binária)

A seguir apresentamos o código em C da função que realiza uma busca binária em um arranjo ordenado em ordem crescente, tentando localizar um dado elemento.

```
28 bool buscaBinaria(int a[], int tam, int alvo, int *pPos){
29   int inicio = 0; // índice do inicio da parte analisada
30   int meio = 0;  // índice do meio da parte analisada
31   int fim = 0  ; // índice do final da parte analisada
32   bool encontrado = false;
33
34   inicio = 0;
35   fim = tam - 1;
36   while (inicio <= fim){
37     meio = (inicio + fim) / 2;
38     if (alvo > a[meio])
39       // Examine a metade superior
40       inicio = meio + 1;
41     else if (alvo < a[meio])
42       // Examine a metade inferior
43       fim = meio - 1;
44     else
45       // Elemento encontrado, forçe a saída
46       inicio = fim + 1;
47   } // while
48
49   *pPos = meio;
50   return alvo == a[meio];
51 } // buscaBinaria
```

```
54 int main(void){
55   int i = 0;
56   int pos = 0;
57   int num = 0;
58   int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
59
60   printf("\nArranjo original ordenado:\n");
61   for(i = 0; i < 10; i++)
62     printf("%d ",a[i]);
63   printf("\n");
64
65   printf("\nDigite o número a ser localizado: ");
66   scanf("%d", &num);
67
68   if (buscaBinaria(a, 10, num, &pos))
69     printf("%d foi encontrado no índice %d\n", num, pos);
70   else
71     printf("%d não foi encontrado no arranjo\n", num);
72
73   return 0;
74 } // main
```

Uso

```
> ./exemplo4.exe

Arranjo original ordenado:
1 2 3 4 5 6 7 8 9 10

Digite o número a ser localizado: 8
8 foi encontrado no índice 7
aula18> ./exemplo4.exe

Arranjo original ordenado:
1 2 3 4 5 6 7 8 9 10

Digite o número a ser localizado: 11
11 não foi encontrado no arranjo
aula18> ./exemplo4.exe

Arranjo original ordenado:
1 2 3 4 5 6 7 8 9 10

Digite o número a ser localizado: 0
0 não foi encontrado no arranjo
```

Pré-processador e Macros

- Durante o processo de compilação pode-se instruir o compilador sobre a maneira como se deseja que ele faça sua tarefa.
- Pode-se fazer isso no próprio código-fonte utilizando os recursos do pré-processador C.
- O pré-processador C é uma ferramenta utilizada antes do estágio de compilação para realizar uma transformação textual no código-fonte.

O pré-processador C

- As instruções dadas para o compilador são definidas a partir das diretivas de pré-processamento.
- O padrão C11 define as seguintes diretivas:

Diretiva	Uso

`#if`
`#ifdef`
`#ifndef`
`#else`
`#elif`
`#endif`
	Compilação condicional

`#include`	Inclusão de arquivos de cabeçalho `.h`
`#define` `#undef`	Definição de constantes ou macros
`#line` `#error`	Diagnóstico
`#pragma`	Informação adicional para o compilador

- Note que todas as diretivas iniciam com `#`.
- Já usamos duas dessas diretivas: `#define`, para definir constantes; e `#include` para incluir arquivos da biblioteca do C.

Incluindo arquivos com #include

- Modularizar e reutilizar códigos são práticas comuns e de grande importância.
- Para isso é necessário instruir o compilador a incluir arquivos-fonte no processo de compilação.
- No pré-processador C essa tarefa é realizada pela diretiva `#include`.
- Por exemplo, dada a linha

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

  o pré-processador a substituirá pelo conteúdo do arquivo de cabeçalho `stdio.h` que contém as definições de funções e macros padrões para a entrada e saída.
- Quando você usa `#include` com os símbolos `<` e `>`, você informa ao compilador para procurar o arquivo nos diretórios do sistema pré-definidos para o compilador.
  - Isso significa que você está se referindo a arquivos de cabeçalho das bibliotecas padronizadas, que já vem com o compilador quando você o instala.

Incluindo arquivos com #include

- Você pode incluir os arquivos de cabeçalhos feitos por você com a mesma diretiva:

```
#include "meu_arquivo.h"
```

- Note o uso das aspas aqui. Ela é necessária para informar que a localização do arquivo de cabeçalho é o diretório de trabalho atual e não o do sistema.
- Ao incluir um arquivo de cabeçalho deve-se cuidar para não incluí-lo mais que uma vez no mesmo arquivo.
  - Se isto acontecer haverá um erro de compilação devido à duplicação de definições.
- Para evitar os erros que podem ocorrer com a inclusão de arquivos, adicionamos uma diretiva de compilação condicional chamada guarda de cabeçalho.
- Basicamente esse mecanismo utiliza uma macro para verificar se o arquivo já foi incluído.
  - Uma prática comum é utilizar o nome do arquivo na macro que será definida, como veremos no exemplo a seguir.

Exemplo 5 (Buscas em arranjos)

Como as operações de busca em arranjo são muito utilizadas, vamos colocá-las em um arquivo de nome `busca.c` em seguida, escrevemos um arquivo de cabeçalho `busca.h` usando a ideia de guarda de cabeçalho. Também será mostrado como usar a nova biblioteca em um programa.

O arquivo de cabeçalho

```
1  #ifndef BUSCA_H
2  #define BUSCA_H
3
4  /**
5   * @file busca.h
6   * @brief Implementa a busca sequencial em arranjos não ordenados e a busca
7   *        binária para arranjos ordenados.
8   *
9   * @author Alexsandro Santos Soares
10  * @date 6/05/2018
11  * @bugs Nenhum conhecido.
12  */
13 #include <stdbool.h>
```

O arquivo de cabeçalho

```
15 /**
16  * @brief Localiza um elemento alvo em um arranjo não ordenado.
17  *
18  * @param[in] a   arranjo com dados
19  * @param[in] tam número de elementos no arranjo
20  * @param[in] alvo elemento procurado
21  * @param[out] pPos endereço onde ficará o índice no arranjo do
22  *                  elemento procurado
23  * @return true, se o elemento for localizado no arranjo.
24  *         false, caso contrário.
25  *
26  * @post caso a busca seja bem sucedida, pos conterá o índice do
27  *       elemento procurado.
28  *       Em caso de falha, pos conterá o índice do último elemento
29  *       do arranjo.
30  */
31 bool buscaSequencial(int a[], int tam, int alvo, int *pPos);
```

O arquivo de cabeçalho

```
34 /**
35  * @brief Localiza um elemento alvo em um arranjo ordenado.
36  *
37  * @param[in] a   arranjo com dados
38  * @param[in] tam número de elementos no arranjo
39  * @param[in] alvo elemento procurado
40  * @param[out] pPos endereço onde ficará o índice no arranjo do
41  *                  elemento procurado
42  * @return true, se o elemento for localizado no arranjo.
43  *         false, caso contrário.
44  *
45  * @post caso a busca seja bem sucedida, pos conterá o índice do
46  *       elemento procurado.
47  *       Em caso de falha, pos conterá o índice do último elemento
48  *       do arranjo.
49  */
50 bool buscaBinaria(int a[], int tam, int alvo, int *pPos);
51
52 #endif
```

Guardas de cabeçalho

No arquivo `busca.h` note o uso do padrão

```
#ifndef BUSCA_H
#define BUSCA_H

texto a ser protegido

#endif
```

Sempre é necessário terminar um `#ifndef` com um `#endif`.
Na primeira vez que o arquivo for incluído a macro `BUSCA_H` ainda não terá sido definida e em consequência disso:

- o conteúdo do arquivo de cabeçalho será copiado para o arquivo que o incluiu.
- A macro `BUSCA_H` será definida.

A partir daí, se tentarmos incluir `busca.h` novamente no mesmo arquivo, não conseguiremos, já que a macro `BUSCA_H` estará definida.

O arquivo busca.c

```
10 #include "busca.h"
```

Observe a inclusão do arquivo de cabeçalho no arquivo de código. Isso é uma boa prática pois o compilador pode verificar quaisquer inconsistências entre eles.

```
28 bool buscaSequencial(int a[], int tam, int alvo, int *pPos){
29   int i = 0; // índice do arranjo
30   bool encontrado = false;
31
32   i = 0;
33   while (i < tam && alvo != a[i])
34     i++;
35
36   *pPos = i;
37   encontrado = (alvo == a[i]);
38
39   return encontrado;
40 } // buscaSequencial
```

O arquivo busca.c

```
60 bool buscaBinaria(int a[], int tam, int alvo, int *pPos){
61   int inicio = 0; // índice do inicio da parte analisada
62   int meio = 0;  // índice do meio da parte analisada
63   int fim = 0  ; // índice do final da parte analisada
64   bool encontrado = false;
65
66   inicio = 0;
67   fim = tam - 1;
68   while (inicio <= fim){
69     meio = (inicio + fim) / 2;
70     if (alvo > a[meio])
71       // Examine a metade superior
72       inicio = meio + 1;
73     else if (alvo < a[meio])
74       // Examine a metade inferior
75       fim = meio - 1;
76     else
77       // Elemento encontrado, forçe a saída
78       inicio = fim + 1;
79   } // while
80
81   *pPos = meio;
82   return alvo == a[meio];
83 } // buscaBinaria
```

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

#include "busca.h"

int main(void){
  int i = 0;
  int pos = 0;
  int num = 0;
  int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  printf("\nArranjo original ordenado:\n");
  for(i = 0; i < 10; i++)
    printf("%d ",a[i]);
  printf("\n");

  printf("\nDigite o número a ser localizado: ");
  scanf("%d", &num);
  if (buscaBinaria(a, 10, num, &pos))
    printf("%d foi encontrado no índice %d\n", num, pos);
  else
    printf("%d não foi encontrado no arranjo\n", num);
  return 0;
} // main
```

Para usar as funções de busca, incluimos o arquivo `busca.h`, como nesse código do arquivo `exemplo5.c`

Compilação separada e uso

Como temos vários arquivos devemos compilá-los separadamente e depois unir os arquivos compilados para formar o programa executável:

```
> gcc -std=c11 -c busca.c
> gcc -std=c11 -c exemplo5.c
> gcc -std=c11 busca.o exemplo5.o -o exemplo5.exe
```

Para usar o programa, digitamos:

```
> ./exemplo5.exe

Arranjo original ordenado:
1 2 3 4 5 6 7 8 9 10

Digite o número a ser localizado: 3
3 foi encontrado no índice 2
```

Usando #define para criar macros

- Já vimos que podemos usar `#define` para definir constantes.
- Agora um outro uso para `#define`: a criação de macros.
- Uma macro é um fragmento de código ao qual é dado um nome.
- Pode-se usar esse fragmento em qualquer parte do programa utilizando o nome dado.
- Por exemplo, podemos definir uma macro que funciona como uma chamada de função:

```
#define areaCirculo(r) (3.1415*r*r)
```

cada vez que o compilador encontrar `areaCirculo(argumento)` ele o substituirá por `(3.1415*(argumento)*(argumento))`, como em `areaCirculo(5)` que é expandido para `(3.1415*5*5)`.

Exemplo 6 (Usando macros)

Nesse exemplo veremos como definir e usar macros no código.

O arquivo exemplo6.c

```
 9 #include <stdio.h>
10
11 #define PI 3.1415
12 #define areaCirculo(r) (PI*r*r)
13
14 int main(void){
15   int raio = 0;
16   float area = 0.0;
17
18   printf("\nDigite o raio: ");
19   scanf("%d", &raio);
20
21   area = areaCirculo(raio);
22   printf("Área = %.2f\n", area);
23
24   return 0;
25 } // main
```

Repare que podemos usar uma macro previamente definida na definição de outra, como foi o caso aqui de `PI` dentro de `areaCirculo`.

Uso

A execução do programa anterior produz:

```
> ./exemplo6.exe

Digite o raio: 5
Área = 78.54
```

Macros pré-definidas

- Existem algumas macros pré-definidas e prontas para o uso em programa.
- Algumas delas são mostradas abaixo.

Macro pré-definida	Valor
`__DATE__`	String contendo o data atual.
`__FILE__`	String conteno o nome do arquivo.
`__LINE__`	Inteiro representando o número atual da linha onde a macro foi posta.
`__STDC__`	Se o compilador segue a padronização ANSI C, então o valor é um inteiro diferente de zero.
`__TIME__`	String contendo o horário atual.

- No próximo slide veremos um exemplo usando todas elas.

Exemplo 7 (Usando macros pré-definidas)

Nesse exemplo veremos como usar macros pré-definidas em um programa.

```
9  #include <stdio.h>
10
11 int main(void){
12
13   printf("\nHoje é%s\n", __DATE__);
14   printf("O nome do arquivo é%s\n", __FILE__);
15   printf("A linha atual é%d\n", __LINE__);
16
17   if (__STDC__ != 0)
18     printf("Este compilador segue o padrão ANSI C\n");
19   else
20     printf("Este compilador não segue o padrão ANSI C\n");
21
22   printf("Horário atual: %s\n", __TIME__);
23
24   return 0;
25 } // main
```

Uso

A execução do programa anterior produz:

```
> ./exemplo7.exe

Hoje é Jul  9 2018
O nome do arquivo é exemplo7.c
A linha atual é 15
Este compilador segue o padrão ANSI C
Horário atual: 15:54:54
```

Makefile

- À proporção que um projeto vai sendo desenvolvido muitos arquivos vão sendo criados.
- Alguns arquivos para serem compilados corretamente necessitam que outros arquivos já tenham sido compilados ou necessitam de uma forma especial de compilação.
- Além disso, se não alterarmos o código fonte e o arquivo já tiver sido compilado, não há necessidade de recompilá-lo.
  - Isso pode economizar bastante tempo de desenvolvimento.
- Lembrar todos esses detalhes fica difícil e, muitas vezes, tedioso.
- Para resolver esses problemas foi criado um utilitário chamado make que compila automaticamente programas e bibliotecas do arquivo fonte por meio da leitura de instruções contidas em arquivos denominados **Makefile**, que especificam como obter o programa de destino.
- Você provavelmente já possui o make instalado, mas se não tiver, basta digitar em Ubuntu:

```
sudo apt install build-essential
```

Makefile

- O objetivo de Makefile é definir regras de compilação para projetos de software.
- O programa make interpreta o conteúdo do Makefile e executa as regras lá definidas.
- O texto contido em um Makefile é usado para a compilação, ligação (*linking*), montagem de arquivos de projeto entre outras tarefas como limpeza de arquivos temporários, execução de comandos, etc.
- Vantagens do uso do Makefile:
  - Evita a compilação de arquivos desnecessários.
  - Automatiza tarefas rotineiras como limpeza de vários arquivos criados temporariamente na compilação.
  - Pode ser usado como linguagem geral de script embora seja mais usado para compilação.

Makefile

- Por exemplo, se seu programa utiliza 120 bibliotecas e você altera apenas uma, o make compara as datas de alteração dos arquivos fontes com as dos arquivos anteriormente compilados e determina qual arquivo foi alterado e compila apenas esse e todos os outros que dependem dele.
- Para ilustrar o uso do make e de seu arquivo Makefile, vamos:
  - Colocar as duas funções de ordenação de arranjos que vimos aqui em uma biblioteca.
  - Usar a biblioteca já criada para busca.
  - Criar um módulo principal.
  - Criar um Makefile para automatizar a tarefa.

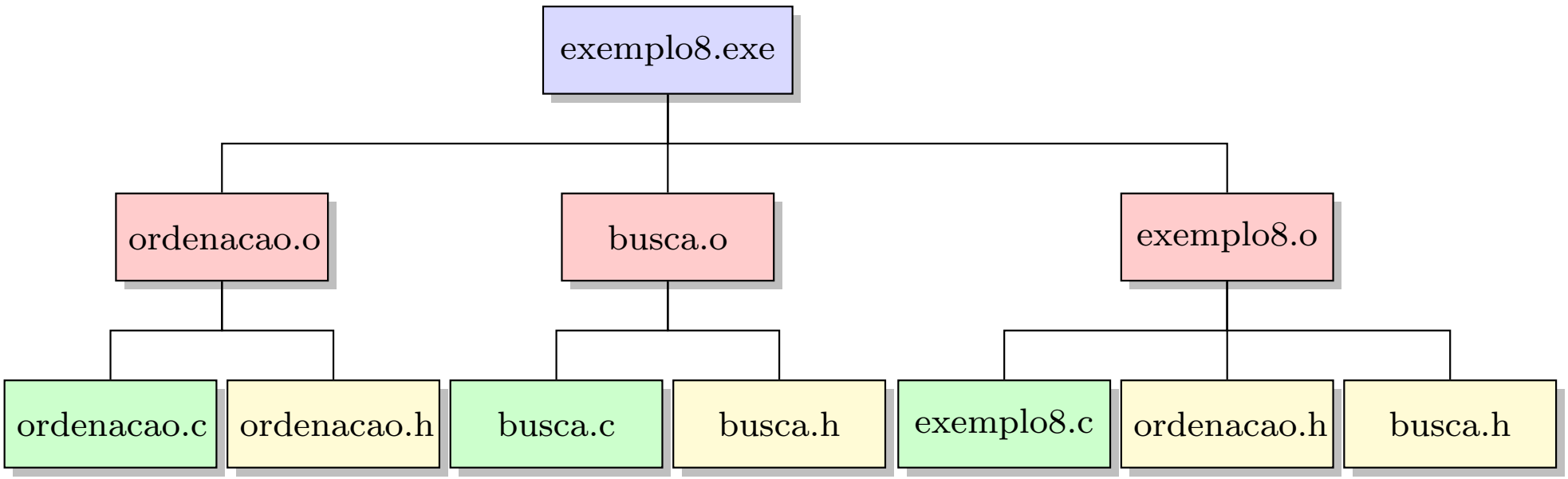
Exemplo de compilação separada

Exemplo 8 (Exemplo de compilação separada)

O programa desse exemplo reune todas as funções de ordenação de arranjos vistas na disciplina na biblioteca `ordenacao.c`. Ele também reúne as funções de busca em arranjos na biblioteca `busca.c`. O módulo principal fica no arquivo `exemplo8.c` que vai chamar uma função da nova biblioteca para ordenar um arranjo e outra de `busca.c` para encontrar um elmento no arranjo ordenado usando busca binária. Para gerenciar o processo de compilação usaremos um arquivo `Makefile`.

Árvore de dependência

Abaixo está a árvore de dependência entre módulos do nosso exemplo.



Essa árvore é criada internamente pelo make a partir das informações no Makefile.

Linhas de dependência

- Um Makefile consiste de linhas de dependência, que definem um destino (uma regra), seguido por dois pontos (:) e opcionalmente um conjunto de arquivos nos quais o destino depende.
- A linha de dependência é organizada de maneira que o destino (do lado esquerdo do caractere :) depende dos componentes (do lado direito do caractere :).
- Após cada linha de dependência, uma série de linhas de texto identado por tabulação pode vir adiante, o que define a maneira de como transformar os componentes (normalmente arquivos fonte) no destino (normalmente a saída).
- Se qualquer um dos componentes for modificado, os comandos abaixo da linha serão executados. A estrutura básica é:

```
# Comentários são iniciados com o símbolo cerquilha (#).
  destino [destino ...]: [componente ...]
  [<TAB>comando 1]
          .
          .
          .
  [<TAB>comando n]
```

O arquivo Makefile

```
1  CC = gcc
2  CFLAGS = -std=c11 -Wall -Wextra -pedantic
3
4  all: exemplo8.exe
5
6  exemplo8.exe: ordenacao.o busca.o exemplo8.o
7    $(CC) $(CFLAGS) ordenacao.o busca.o exemplo8.o -o exemplo8.exe
8
9  ordenacao.o: ordenacao.c ordenacao.h
10   $(CC) $(CFLAGS) -c ordenacao.c
11
12 busca.o: busca.c busca.h
13   $(CC) $(CFLAGS) -c busca.c
14
15 exemplo8.o: exemplo8.c ordenacao.h busca.h
16   $(CC) $(CFLAGS) -c exemplo8.c
17
18 clean:
19   rm -rf *.o exemplo8.exe *~
```

Observações sobre o Makefile

- `CFLAGS` são os opções de compilação que serão passadas para o GCC:

  -Wall liga todas as mensagens de avisos do compilador. Essa opção deve sempre ser usada para gerar um código melhor.

  -Wextra liga mais mensagens extras de avisos que não são ligadas por `-Wall`.

  -pedantic informa todas as mensagens demandas pela padronização ISO C e rejeita todos os programas que usem extensões proibidas ou que não sigam o ISO C informado pela opção `-std`.

Uso do make

```
> make

gcc -std=c11 -Wall -Wextra -pedantic -c ordenacao.c
gcc -std=c11 -Wall -Wextra -pedantic -c busca.c
busca.c: In function ‘buscaBinaria’:
busca.c:64:8: warning: unused variable ‘encontrado’ [-Wunused-variable]
   bool encontrado = false;
        ^~~~~~~~~~
gcc -std=c11 -Wall -Wextra -pedantic -c exemplo8.c
gcc -std=c11 -Wall -Wextra -pedantic ordenacao.o busca.o exemplo8.o -o exem
```

Depois de retirar a linha 68, vamos recompilar o projeto:

```
> make

gcc -std=c11 -Wall -Wextra -pedantic -c busca.c
gcc -std=c11 -Wall -Wextra -pedantic ordenacao.o busca.o exemplo8.o -o exem
```

Note que apenas o código fonte de `busca.c` foi recompilado. Como isso gerou um novo arquivo objeto `busca.o`, o programa `exemplo8.exe` também foi recompilado – na verdade ele só foi ligado.

Uso do programa exemplo8.exe

```
> ./exemplo8.exe

Arranjo original ordenado:
10 8 6 4 5 3 7 9 2 1
Arranjo ordenado:
1 2 3 4 5 6 7 8 9 10

Digite o número a ser localizado: 3
3 foi encontrado no índice 2
```

Funções recursivas

- Uma função que chame a si mesma é uma função recursiva.
- C pode lidar com funções recursivas, embora esse não seja o estilo de programação predileto de quem escolhe essa linguagem.
- Por isso, em geral, a recursão em C é mais lenta que a iteração usando laços.
  - Linguagens mais modernas como Haskell, Ocaml e outras linguagens declarativas usam a recursão como modo básico de repetição.
  - Os compiladores para essas linguagens são melhores equipados para transformar funções recursivas em código mais eficiente.
- Entrentao, a recursão torna o programa mais elegante e mais claro.
- Muitos algoritmos e estruturas de dados são definidas recursivamente e isso os torna mais fáceis de compreender e de provar sua corretude.
- No que segue, veremos alguns exemplos de funções recursivas em C.

Fatorial recursivo

O fatorial de um número é definido por

$$\text{fatorial}(n) = \begin{cases} 1, & \text{se } n = 0 \\ n \times \text{fatorial}(n-1), & \text{se } n > 0 \end{cases}$$

Abaixo está a definição dessa função em C:

```
long int fatorial(int n)
{
   if (n > 0)
      return n * fatorial(n-1);
   else
      return 1;
} // fatorial
```

Podemos chamar essa função da mesma forma que podemos chamar qualquer outra função em C:

```
long int f;
f = fatorial(5);
```

Sequência de Fibonacci

O n-ésimo número de Fibonacci é definido por:

$$\text{fib}(n) = \begin{cases} 0, & \text{se } n = 0 \\ 1, & \text{se } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2), & \text{se } n > 1 \end{cases}$$

Abaixo está a definição dessa função em C:

```
long int fib(int n)
{
   if (n > 1)
      return fib(n-1) + fib(n-2);
   else if (n == 1 || n == 0)
      return n;
} // fib
```

Para saber mais

- Forouzan, B. A and Gilbert, R. F. *Computer Science: a structured programming approach using C*. 3rd edition. Cengage Learning, 2007.
- Alexandre Ramos. *Guia Rápido: GCC, Makefile e Valgrind.*. Disponível em `https://inf.ufes.br/~pdcosta/ensino/2017-1-estruturas-de-dados/material/GuiaRapido_EDI.pdf`.

Fontes

- Forouzan, B. A and Gilbert, R. F. *Computer Science: a structured programming approach using C*. 3rd edition. Cengage Learning, 2007.
- *Pré-processador C – Parte 1*. Disponível em `https://www.embarcados.com.br/pre-processador-c-parte-1/`.
- *C Preprocessor and Macros*. Disponível em `https://www.programiz.com/c-programming/c-preprocessor-macros`.
- *Introdução ao Makefile*. Disponível em `https://www.embarcados.com.br/introducao-ao-makefile/`.
- *Tutorial: Aprenda a criar seu próprio makefile*. Disponível em `https://www.mat.uc.pt/~pedro/lectivos/ProgramacaoOrientadaObjectos1617/tutorialMakefilesPT.pdf`.
- *Programar em C/Makefiles*. Disponível em`https://pt.wikibooks.org/wiki/Programar_em_C/Makefiles`.