Notas de aula de Introdução à Ciência da Computação

Daniel Oliveira Dantas

6 de novembro de 2013

Sumário

Capítulo 1

Introdução ao C

O C é uma linguagem *case sensitive*, compilada, com tipagem estática e fraca. *Case sensitive* porque letras maiúsculas são consideradas diferentes de minúsculas. Seu código não roda interativamente como nas linguagens interpretadas. Deve ser compilado, ou seja, gerar código executável antes de rodar. Se diz que o C tem tipagem estática pois o tipo de uma variável, uma vez definido, não muda mais. Se diz que C tem tipagem fraca pois é possível realizar operações entre variáveis de tipos diferentes, como somar inteiros e reais.

1.1 Palavras reservadas

- Palavras reservadas ou palavras chave: são palavras que não podem ser usadas para nomear variáveis ou funções criadas pelo usuário.

```
char      void     for
double    true     if
float     false    else
int       switch   typedef
long      case     struct
short     default  return
signed    break    main
unsigned  while    sizeof
```

1.2 Um programa em C mínimo

- Um programa em C mínimo: este programa não faz nada, simplesmente declara uma função `main` vazia. A função `main` é o ponto de entrada de qualquer projeto em C, ou seja, por onde o programa começa a execução. Neste exemplo não há nenhuma instrução dentro da `main`.

```
int main(void)
{
```

```
}
```

- Outro programa em C: este programa simplesmente imprime uma mensagem na tela através de uma chamada à função `printf`. Para usar `printf`, é necessário incluir o arquivo de cabeçalho `stdio.h`.

```
#include <stdio.h>
int main(void)
{
    printf("Hello world!\n");
}
```

- Ainda outro programa em C: como no exemplo anterior, imprime uma mensagem na tela. Além disso, espera o usuário pressionar uma tecla antes de terminar a execução através de uma chamada de sistema a um comando chamado `pause`. Essa chamada é feita através da função `system`. Para usar `system`, é necessário incluir o arquivo de cabeçalho `cstdlib`.

```
#include <stdio.h>
#include <cstdlib>
int main(void)
{
    printf("Hello world!\n");
    system("pause");
}
```

1.3 Programando em C

Para programar em C é necessário um compilador. Se seu sistema operacional for Windows, instale o Orwell Dev-C++, conhecido como *devcpp*, disponível em:

`http://sourceforge.net/projects/orwelldevcpp/`

Se seu sistema operacional for Linux, instale o gcc rodando o comando `sudo apt-get install gcc`

- Como criar um programa em C

    1 - Editar o código fonte e salvar em um arquivo com extensão cpp.

    2 - Compilar o código fonte usando um compilador.

    3 - Rodar o programa.

- Usando o devcpp: o devcpp é um ambiente integrado de programação, com editor e compilador na mesma aplicação.

  1 - Criar um novo *source file* com ctrl + n.

  2 - Editar e salvar o arquivo com extensão cpp.

  3 - Pressionar F9 para compilar e rodar.

  4 - Ler as mensagens do compilador e corrigir os erros.

  5 - *Go to* 3.

- Usando o gcc: o gcc é um compilador de linha de comando.

  1 - Criar um novo *source file* com um editor de texto como o emacs.

  2 - Editar e salvar o arquivo com extensão cpp.

  3 - Compilar usando o gcc. Antes de rodar a linha de comando abaixo, substitua o conteúdo entre `<>` pelos nomes dos arquivos.

  ```
  gcc <nome_arq_fonte.cpp> -o <nome_arq_executável>
  ```

  4 - Ler as mensagens do compilador e corrigir os erros de compilação.

  5 - Chamar o executável caso tenha compilado e corrigir os erros de lógica.

  6 - *Go to* 3.

1.4 Literais

Literais são valores com os quais podemos trabalhar. Podem ser de diversos tipos:

```
Números inteiros:       0       Números reais:     2.0
                        -1                         3.1415
                        333                        -1000.0

Inteiros hexadecimais:  0xff    Inteiros octais:   077
                        0x002a                     06
                        0xd                        02342

Caracteres:             'a'     Strings:           "Hello world"
                        '4'                        "bom dia"
                        '\n'                       "Idade = %d anos"

Valores booleano:       true
                        false
```

1.5 Chamada de funções

- Funções são trechos de código encapsulados que executam alguma operação.
- Funções são chamadas pelo nome.
- Recebem zero ou mais parâmetros entre parênteses. O número de parâmetros é definido na sua criação e pode ser constante (as funções `sin`, `cos`, `log` recebem um parâmetro; a função `pow` recebe dois parâmetros) ou variável (`printf` recebe um ou mais parâmetros).
- Podem retornar um valor de algum tipo de dado. Esse tipo (real, inteiro, caracter etc) é definido no momento da sua criação. O valor retornado pode mudar dependendo dos parâmetros que a função recebe ao ser chamada.
- Podem também não retornar nada. São as funções de tipo `void`.
- Exemplos de chamadas:

```
printf("Hello\n");
printf("O mundo termina no ano de %d\n", 2012);
sin(1.0);
pow(2.0, 3.0);
printf("O seno de pi/2 eh %f\n", sin(3.14/2.0));
```

1.6 A biblioteca de funções matemáticas

Para usá-la, adicione a linha seguinte ao início do código:

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

- Caso esteja usando Linux, adicionar a opção `-lm` à linha de comando de chamada ao gcc.
- Contém funções como

```
sin   cos    tan
asin  acos   atan
log   log10  pow
ceil  floor  fabs
```

- Contém constantes como

```
M_PI  M_E
```

- Abaixo está um exemplo de programa que usa essa biblioteca. O programa imprime uma mensagem com o valor do seno de de pi/4 através de uma chamada a `printf`, e espera o usuário pressionar uma tecla antes de finalizar.

```
#include <stdio.h>
#include <cstdlib>
#include <math.h>
int main(void)
{
    printf("O seno de pi/4 eh %f\n", sin(3.14/4.0));
    system("pause");
}
```

1.7 Comentários

Comentários são textos ignorados pelo compilador. Dentro de um comentário é possível escrever qualquer coisa, sem risco de erros de compilação. Em C existem dois tipos de comentário:

- Comentários de uma linha: começa com `//` e vai até o final da linha.
- Comentários de múltiplas linhas: começa com `/*` e termina com `*/`.
- Exemplo de programa com comentários.

```
/*
Programa que imprime o valor do seno
de 45 graus, ou pi/4.
*/
#include <stdio.h> // biblioteca do printf
#include <cstdlib> // biblioteca do system
#include <math.h>  // biblioteca do sin
int main(void)
{
    printf("O seno de pi/4 eh %f\n", sin(3.14/4.0));
    system("pause");
}
```

1.8 Identificadores

Identificadores são os nomes de variáveis, funções, constantes e outros objetos, tanto os criados pelo usuário quanto os contidos em bibliotecas.

- O primeiro caracter pode ser letra ou sublinhado
- Os demais caracteres podem ser letras, dígitos ou sublinhado.
- Não pode ser igual a nenhuma palavra reservada.
- Exemplos:

Corretos:	_ABC	Incorretos:	123
	contador		i+1
	i		a!
	x2		r$

1.9 Variáveis

Variáveis são posições de memória com um nome, usadas para armazenar valores. Os valores armazenados podem ser modificados a qualquer momento.

- O nome de uma variável deve ser um identificador válido, como descrito na Seção 1.8.
- Sua declaração e inicialização devem aparecer no código antes de seu primeiro uso.
- A declaração de uma variável é onde definimos seu tipo e seu identificador, isto é, seu nome.
- A inicialização de uma variável é onde atribuímos a ela algum valor.
- A declaração de uma variável tem o seguinte formato:

```
<tipo de dados> <identificador>;
```

- Exemplos de declarações de variáveis:

```
int x;
int i;
float f;
char c;
unsigned int file_size;
unsigned char idade;
unsigned short int ano;
double p1;
long long int n;
long double value;
```

- Variáveis podem ser:

  Locais, quando declaradas dentro de uma função.

  Globais, quando declaradas fora de funções.

  Parâmetros, quando declaradas na lista de parâmetros da definição de uma função.

- Variáveis locais e globais podem ser inicializadas, ou seja, receber algum valor, no momento de sua declaração. O formato de uma declaração com inicialização é o seguinte:

  ```
  <tipo de dados> <identificador> = <valor>;
  ```

  Lembrando que identificador é o nome. O valor pode ser um literal, como descrito na Seção 1.4, outra variável, uma chamada de função que retorna algum valor, ou mesmo uma expressão matemática.

- Exemplos de declarações de variáveis com inicialização:

```
int x = 1;
int i = 0;
float e = 2.7;
char c = 'a';
double p = sin(M_PI/4.0);
double p2 = p;
double p_dup = 2.0 * p;
```

1.10 Constantes

Como variáveis, constantes são posições de memória que armazenam algum valor. Porém ao contrário de variáveis, o valor não pode ser modificado.

- A declaração de uma constante tem o seguinte formato:

  ```
  const <tipo de dados> <identificador> = <valor>;
  ```

- Exemplos de declarações de constantes:

```
const long int c = 299792458;
const float pi = 3.141592653589793238462;
const float G = 6.67428E-11;
const double h = 6.62606896E-34;
```

1.11 Tipos de dados

Em um computador digital, todo e qualquer dado é armazenado em forma de bits. O conteúdo de uma posição de memória pode ser interpretado como um número inteiro, real ou até mesmo uma string. Portanto, para que um dado seja interpretado corretamente, é necessário associar a esse dado o seu tipo. Na linguagem C os tipos mais importantes são:

- `int`: valor inteiro. Normalmente ocupa 4 bytes, sendo capaz de representar inteiros de -2147483648 a 2147483647.
- `float`: valor real. Normalmente ocupa 4 bytes, sendo capaz de representar valores reais com cerca de 8 dígitos significativos.
- `double`: valor real. Normalmente ocupa 8 bytes, sendo capaz de representar valores reais com cerca de 16 dígitos significativos.
- `char`: valor inteiro pequeno ou caracter. Ocupa 1 byte, sendo capaz de representar inteiros de -128 a 127.
- `bool`: valor lógico. Pode ser `true` ou `false`.
- `void`: tipo vazio. Usado para declarar funções que não retornam valor ou ponteiros sem tipo associado.

Alguns tipos de dados podem ser alterados por modificadores:

- `unsigned`: pode modificar os tipos `int` e `char`. Uma variável `unsigned` é sempre positiva, e pode representar valores duas vezes maiores que sua correspondente `signed`
- `signed`: pode modificar os tipos `int` e `char`. Por padrão toda variável `int` e `char` é `signed`, ou seja, com sinal.
- `short`: pode modificar o tipo `int`, diminuindo sua capacidade. Normalmente, em máquinas de 64 bits, um short int ocupa 2 bytes.
- `long`: pode modificar os tipos `int` e `double`, aumentando sua capacidade. Normalmente, em máquinas de 64 bits, um long int ocupa 8 bytes e um long double, 16 bytes.

Para descobrir quanto espaço ocupa uma variável de um certo tipo de dados, usar o operador unário `sizeof`. `sizeof` pode ser usado com tipos de dados ou com variáveis. No primeiro caso, o uso de parênteses é obrigatório, e no segundo, é opcional. Observe o exemplo abaixo.

```
#include <stdio.h>
#include <cstdlib>
int main(void)
{
```

```
    char C;
    printf("Um int ocupa %ld bytes\n", sizeof(int));
    printf("A variavel C ocupa %ld bytes\n", sizeof(C));
    system("pause");
}
```

1.12 Expressões numéricas

Expressões numéricas são compostas por operadores e operandos. Os operandos podem ser literais, funções que retornam algum valor, variáveis, constantes, ou mesmo outras expressões. Os operadores podem ser de atribuição, de comparação, aritméticos, lógicos ou bit a bit.

- Uma expressão elementar contém apenas um operando:

  `<literal>`

  `<função que retorna valor>`

  `<variável>`

  `<constante>`

- Exemplos de expressões elementares:

  10
  sin(M_PI)
  x
  M_PI

- Uma expressão composta pode conter vários operadores e operandos:

  `( <expressão> <operador binário> <expressão> )`

  `( <operador unário> <expressão> )`

- Exemplos de expressões compostas:

  2 + 2
  −1
  x = 10
  sin(M_PI) + 1.0
  y = (3 + x)
  n = 2 * (3 + i)

1.12.1 O operador de atribuição =

É usado para atribuir um valor a uma variável. É importante salientar que, antes de usar uma variável, ela deve ser inicializada, ou seja, algum valor deve ser atribuído a essa variável. Uma variável não inicializada pode conter um valor imprevisível.

- A forma geral de uma atribuição é a seguinte:

  `<identificador da variável> = <expressão>;`

  Onde é claro que `expressão` pode ser elementar, como um valor literal, outra variável, uma constante, uma chamada a função; ou composta, como uma operação aritmética, lógica etc.

- Exemplos de atribuições:

```
n = 10;
m = n;
y = -(3 + x);
n = 3 * (4 + i);
```

1.12.2 Operadores aritméticos

Expressões com operadores aritméticos retornam valores numéricos, inteiros ou reais.

Precedência:		
maior	`++ --`	(incremento e decremento unário)
	`-`	(menos unário)
	`* / %`	(multiplicaćão, divisão e módulo)
menor	`+ -`	(soma e subtração)

- Operadores de maior precedência são avaliados antes dos de menor precedência.
- Operadores com mesmo nível de precedência são avaliados da esquerda para a direita.
- É possível usar parênteses para modificar a ordem das operações
- `i++` é o mesmo que `i = i + 1`.
- `i--` é o mesmo que `i = i - 1`.
- Nos exemplos abaixo, que valor é atribuído à variável à esquerda do operador `=`?

```
a = 2 * 3 + 1;
b = 1 + 2 * 3;
c = 10 / 2 * 5;
d = 10 % 3;
e = 2 * 10 % 4;
f = 3 * (2 + 1);
g = 2;
h = 3 * ++g;
i = 3 * g++;
```

1.12.3 Operadores lógicos e de comparação

Expressões com operadores lógicos ou de comparação retornam valores do tipo `bool`, ou seja, `true` ou `false`.

Precedência:		
maior	`!`	(negação lógica)
	`> >= < <=`	(desigualdade)
	`== !=`	(igualdade e diferença)
	`&&`	(and lógico)
menor	`||`	(or lógico)

- Nos exemplos abaixo, que valor é atribuído à variável à esquerda do operador `=`?

```
a = 1 < 2;
b = true || false;
c = true && false;
d = 1 < 2 || 4 < 2;
e = 1 < 2 && 4 < 2;
f = 1 < 1;
g = !(3 < 2);
e = 3 == 4;
e = 3 != 4;
```

- Para testar se o valor de uma variável está em um certo intervalo, a construção é a seguinte:

```
f > 0 && f < 1
```

  para testar se f está entre 0 e 1. Jamais omitir o `&&` como na expressão INCORRETA abaixo:

$0 < f < 1$

1.12.4 Operadores bit a bit

Operam sobre o valor binário de uma variável. A variável só pode ser do tipo `int`, `char` ou suas variantes.

Precedência:		
maior	`~`	(complemento de um)
	`<<  >>`	(shift left e shift right)
	`&`	(and binário)
	`^`	(xor binário)
menor	`|`	(or binário)

- Nos exemplos abaixo, que valor é atribuído à variável à esquerda do operador `=`?

```
a = 3 | 1;
b = 1 >> 1;
c = 1 << 1;
d = 1 << 2;
e = 5 ^ 3;
f = ~ 0;
```

Capítulo 2

Controle de fluxo

Um programa de computador é uma sequência de instruções. Normalmente as instruções são executadas na ordem em que aparecem no programa. Porém, isso permite apenas a criação de programas muito simples.

Imagine que tenhamos por exemplo uma matriz com centenas de valores. Para processar cada item dessa matriz, precisamos executar centenas de instruções. Se a instrução for a mesma para todos os itens da matriz, podemos facilitar nossa vida criando um programa bem mais simples se colocarmos essa instrução dentro de um *laço*, ao invés de ter uma cópia desta mesma instrução para cada item da matriz.

Outro caso em que o controle de fluxo é útil é quando uma instrução é executada em apenas alguns casos. Imagine que tempos um banco de dados com milhões de valores de CPF e queremos imprimir apenas o nome do portador de um certo CPF. Podemos usar uma instrução do tipo `if` que, para cada CPF lido, testa se é igual ao CPF desejado e imprime o nome do portador somente em caso afirmativo.

2.1 if / else

Uma instrução do tipo `if` executa um bloco de instruções somente se uma dada condição é verdadeira. Opcionalmente pode ser complementada com uma instrução `else`, que executa outro bloco caso a dada condição seja falsa.

- A forma geral de uma instrução `if` é a seguinte:

```
if (<expressão condicional>)
{
      <bloco executado se a condição for verdadeira>
}
```

- A forma geral de uma instrução `if`/`else` é a seguinte:

```
if (<expressão condicional>)
{
     <bloco executado se a condição for verdadeira>
}
else
{
     <bloco executado se a condição for falsa>
}
```

- Observe que a expressão condicional só aparece uma vez, junto ao `if`, jamais junto ao `else`.
- A expressão condicional pode ser uma expressão lógica, de comparação, ou uma operação aritmética com inteiros. Nao usar expressões aritméticas com `float` ou `double`.
- O bloco do `if` é executado se a expressão condicional der como resultado `true` ou diferente de zero.
- O bloco do `else` é executado se a expressão condicional der como resultado `false` ou igual a zero.

2.2 for

Palavra reservada para criar instruções de iteração, ou seja, laços. Um laço é um bloco que é executado várias vezes.

- A forma geral de uma instrução `for` é a seguinte:

```
for (<inicialização>; <condição de continuidade>; <incremento>)
{
     <bloco a ser executado várias vezes>
}
```

- Na inicialização, geralmente uma variável de controle é declarada e inicializada.
- Variável de controle é a que indica quando o laço deve parar.
- Na condição de continuidade, se testa se o laço deve continuar. Geralmente isso é feito comparando a variável de controle com algum valor.
- No incremento, a variável de controle é incrementada. Pode também ser decrementada se necessário.

2.3 while

Faz exatamente o mesmo que o `for`, ou seja, serve para criar laços. A diferença está na sintaxe.

- A forma geral de uma instrução `while` é a seguinte:

```
<inicialização>;
while (<condição de continuidade>)
{
      <bloco a ser executado várias vezes>
      <incremento>;
}
```

- A inicialização, condição de continuidade e incremento funcionam exatamente da mesma maneira que no `for`, conforme explicado na Seção 2.2

Capítulo 3

Indentação

Indentação é a maneira como avançamos ou recuamos as instruções no texto do código fonte para facilitar o entendimento. A linguagem C ignora completamente a indentação. Porém, para o programador, enquanto um código mal indentado pode aparecer completamente ilegível, o mesmo código bem indentado costuma aparecer claro e compreensível.

Observe o código abaixo:

```
#include <stdio.h>
int main (void){long int val=1;for(int n=1;
n<=8;n++){val=val*n;}printf("val = %ld\n",val);}
```

Esse código possui exatamente as mesmas instruções que o código a seguir. A única diferença está na indentação. Observe como o código abaixo é mais legível:

```
#include <stdio.h>
int main (void)
{
  long int val = 1;
  for (int n = 1; n <= 8; n++)
  {
    val = val * n;
  }
  printf("val = %ld\n", val);
}
```

Em C, *bloco* se refere a linhas de código entre chaves. Abrimos um novo bloco de código quando declaramos uma função, quando criamos uma instrução de laço com `for` ou `while`, ou uma instrução de controle de fluxo com `if` ou `switch`. Para se indentar um código em C como no exemplo acima, siga as instruções a seguir:

- A chave que abre um bloco fica abaixo da primeira letra da instrução que abre o bloco.
- Cada vez que se abre uma chave, avançar 4 espaços. Cada vez que se fecha uma chave, recuar 4 espaços.
- A chave que fecha um bloco fica exatamente abaixo da chave que abre o bloco.
- Cada vez que se abre uma chave, pular uma linha para que a primeira instrução do bloco fique na linha seguinte à da chave.

Todo código deve ser indentado corretamente, tanto código escrito em papel quanto em arquivos digitais.

Capítulo 4

Entrada e saída

Entrada e saída se refere à comunicação do programa, tanto com o usuário quanto com arquivos em disco. Duas funções que podem ser usadas para fazer entrada e saída com o usuário pelo console são `printf` e `scanf`. Para fazer entrada e saída com arquivos, pode-se usar `fprintf` e `fscanf`. Para usar qualquer uma dessas funções, adicionar a seguinte linha ao início do código:

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

4.1 printf

`printf` não é uma palavra reservada como `while`, e sim uma função. `printf` espera como primeiro parâmetro uma string, como nos exemplos abaixo.

```
printf("Hello world\n");
printf("bom dia");
printf("Pressione enter para continuar...\n");
```

Observe que strings ficam entre aspas duplas. Ao final de algumas das strings acima está o caractere especial '\n'. Esse caractere representa um pulo para a linha seguinte. Sempre adicionar esse caractere ao final quando a string contiver uma mensagem que deve aparecer isolada em uma linha.

Além do pulo de linha, existem outros caracteres especiais que `printf` reconhece, como pode ser visto na Tabela 4.1

Caractere	Valor ASCII	Representação
Pula linha	10	`\n`
Tabulação	9	`\t`
Beep	7	`\a`
Backslash	92	`\\`
Aspas duplas	34	`\"`
Símbolo de percentagem	37	`%%`

Tabela 4.1: Caracteres especiais do C

Para ser mais versátil, `printf` pode imprimir valores de expressões no meio da mensagem. Para isso é necessário colocar na string um especificador de formato no ponto em que o valor deve aparecer, e adicionar como parâmetro a expressão a ser impressa. Especificadores de formato são precedidos pelo símbolo % e servem para definir como interpretar o valor da expressão, ou seja, se é um valor inteiro, real, caractere etc. Observe o exemplo abaixo.

```
printf("Um inteiro ocupa %d bytes\n", sizeof(int));
printf("PI vale %f\n", 3.141592653589793238462);
```

Cada uma das strings possui um especificador de formato. A primeira possui um `%d` para imprimir um número inteiro e a segunda possui um `%f` para imprimir um número real. A saída do exemplo acima será a seguinte:

```
Um inteiro ocupa 4 bytes
PI vale 3.141593
```

Por padrão o `%f` imprime o número real com seis casas decimais, mas esse comportamento pode ser modificado colocando logo depois do símbolo `%` um ponto seguido do número de casas decimais desejado, como a seguir.

```
printf("PI vale %.10f\n", 3.141592653589793238462);
```

cuja saída será

```
PI vale 3.1415926536
```

O formato de números inteiros também pode ser modificado. Colocando um número `n` de dígitos entre o `%` e o `d`, fará cada inteiro ocupar `n` espaços. Isso é útil para se imprimir tabelas onde o dígito menos significativo deve ficar alinhado à direita. Para completar o inteiro com zeros à esquerda, colocar um zero entre o `%` e o número de dígitos completará o número impresso com zeros à esquerda até que ocupe o tamanho desejado. Observe os exemplos abaixo e compare suas saídas.

Saída	Especificador	Exemplo
Inteiro decimal	%d	45
Inteiro decimal `long int`	%ld	12345678901234
Inteiro hexadecimal	%x	f3 ou F3
	%X	f3 ou F3
Número real `float`	%f	3.141593
Número real `double`	%lf	3.141593
Número real em notação científica	%e	1.000000e+09
	%E	1.000000E+09
Caractere	%c	a
String	%s	Hello
Endereço de ponteiro	%p	0x7fffec2e33a8

Tabela 4.2: Especificadores de formato do C

```
printf("Um inteiro ocupa %d bytes\n", sizeof(int));
printf("Um inteiro ocupa %6d bytes\n", sizeof(int));
printf("Um inteiro ocupa %06d bytes\n", sizeof(int));
```

terão como saída respectivamente

```
Um inteiro ocupa 4 bytes
Um inteiro ocupa      4 bytes
Um inteiro ocupa 000004 bytes
```

Em uma mesma string pode haver vários especificadores de formato, bastando que, para cada um dos especificadores, deve haver um valor correspondente como parâmetro. Observe o exemplo abaixo. Existem dois especificadores de formato. O primeiro será substituído pelo valor de `val1` e o segundo, pelo valor de `val2`.

```
int val1 = 33;
float val2 = 1.23;
printf("A variavel inteira vale %d e a variavel real
    vale %f\n", val1, val2);
```

4.2 scanf

A chamada a `scanf` é semelhante à chamada ao `printf`. O primeiro parâmetro também é uma string, mas deve possuir um especificador de formato. Os demais parãmetros são os endereços das posições de memória onde os dados serão gravados. É recomendável porém usar uma chamada `scanf` para cada valor a ser lido. Para passar o endereço de memória, é necessário preceder a variável pelo símbolo `%`.

```
int i;
scanf("%d", &i);
float x;
scanf("%f", &x);
char letra;
scanf("%c", &letra);
```

- `scanf` retorna o número de valores lidos com sucesso. Idealmente deve-se usar esse valor para testar se a leitura foi bem sucedida.
- Segue um exemplo de código com `scanf`.

```
#include <stdio.h>
#include <cstdlib>
int main (void)
{
  int result;
  float userfloat;
  printf("Digite um numero real:\n");
  result = scanf("%f", &userfloat);
  printf("result = %d, real digitado = %f\n",
      result, userfloat);
  system("pause");
}
```

- No exemplo acima, `userfloat` armazena o real digitado pelo usuário. A variável `result` será igual a 1 caso a leitura seja bem sucedida, e 0 caso contrário.

4.3 sscanf

`scanf` pode não funcionar corretamente quando o usuário digita valores de tipos diferentes do esperado. Uma maneira de contornar isso é armazenando a string em uma variável intermediária antes de fazer a leitura. Nesse caso, a leitura é feita com `sscanf`, uma versão alternativa de `scanf` que lê dados de strings.

```
#include <stdio.h>
#include <cstdlib>
int main (void)
{
  int result;
  float userfloat;
  char line[256];
```

```
  printf("Digite um numero real:\n");
  gets(line);
  result = sscanf(line, "%f", &userfloat);
  printf("result = %d, real digitado = %f\n", result,
      userfloat);
  system("pause");
}
```

- No exemplo acima, podemos observar uma nova variável chamada `line`, que armazenará o texto digitado pelo usuário. `gets` lê o texto digitado e o armazena em `line`, que é passado como parâmetro para `sscanf`. As demais partes do código permanecem como no exemplo da Seção 4.2.

4.4 fscanf

`fscanf`é uma função que pode ser usada para leitura de dados de arquivos. Recebe um parâmetro a mais que `scanf`: o ponteiro para o arqivo a ser lido.

- Para obter o ponteiro para o arquivo, usar a função `fopen`. No exemplo abaixo, `fp` é a variável que armazenará o ponteiro para o arquivo. O primeiro parâmetro é uma string que contém o nome do arquivo a ser aberto. Neste exemplo é `arquivo.txt` mas poderia ser qualquer outro. O segundo parâmetro indica o tipo de acesso desejado, ou seja, leitura ou escrita. Nesse caso contém uma letra `r` para indicar que o arqivo será aberto para leitura. Usar a letra `w` para escrita. `fopen` retorna zero em caso de erro, ou seja, se o arquivo não existir ou se não houver permissão para o acesso desejado.

  ```
  FILE* fp = fopen("arquivo.txt", "r");
  ```

- Uma maneira de testar se o arquivo foi aberto corretamente é mostrada a sqguir. Uma mensagem mais esclarecedora para o usuário, e portanto mais útil, pode incluir o nome do arquivo.

  ```
  if (fp == 0)
  {
    printf("Erro ao abrir arquivo\n");
    return 1;
  }
  ```

- Uma vez que o arquivo esteja aberto, chamar `fscanf` com o ponteiro retornado por `fopen` como primeiro parâmetro. Como `scanf`, recebe também uma string com o especificador de formato, e a variável onde o valor lido deve ser armazenado. Essa variável deve ser passada por referência, ou seja, é necessário usar o símbolo `&` antes do identificador da variável para indicar que é passado seu endereço como parâmetro.

```
int i;
fscanf(fp, "%d", &i);
float x;
fscanf(fp, "%f", &x);
char letra;
fscanf(fp, "%c", &letra);
```

- Segue um programa exemplo que lê dois números inteiros de um arquivo. Cada número deve estar em uma linha diferente no arquivo.

```
#include <stdio.h>
int main (void)
{
  int a, b;
  FILE* fp = fopen("arquivo.txt", "r");
  if (fp == 0)
  {
    printf("Erro ao abrir arquivo\n");
    return 1;
  }
  fscanf(fp, "%d", &a);
  printf("Valor de a = %d\n", a);
  fscanf(fp, "%d", &b);
  printf("Valor de b = %d\n", b);
  fclose(fp);
}
```

- Abaixo está uma versão alternativa que usa `fgets` para ler cada linha do arquivo por completo antes de tentar extrair dela o número. Observe que a função `fgets` é usada no lugar da `gets`, da Seção 4.3.

```
#include <stdio.h>
int main (void)
{
  int MAX_LEN = 256
  int a, b;
  FILE* fp = fopen("arquivo.txt", "r");
```

```
  if (fp == 0)
  {
    printf("Erro ao abrir arquivo\n");
    return 1;
  }
  char line[MAX_LEN];
  fgets(line, MAX_LEN, fp);
  sscanf(line, "%d", &a);
  printf("Valor de a = %d\n", a);
  fgets(line, MAX_LEN, fp);
  sscanf(line, "%d", &b);
  printf("Valor de b = %d\n", b);
  fclose(fp);
}
```

- Idealmente, deve-se testar se cada `fscanf` e `fgets` fez uma leitura bem sucedida, como no trecho de código abaixo. `fgets` recebe três parâmetros: um vetor de caracteres ou string, o tamanho máximo da string e o ponteiro para o arquivo. Retorna zero em caso de erro.

```
char* result1 = fgets(line, MAX_LEN, fp);
int result2 = sscanf(line, "%d", &n);
if (result1 != 0 && result2 == 1)
{
  printf("Valor lido = %d\n", result, n);
}
else
{
  printf("Valor invalido\n");
}
```

- Ao final do programa, após fazer todas as leituras necessárias, fechar o arquivo com `fclose`, como na linha de código a seguir. Isso salva as escritas no arquivo e o libera para que outros programas possam acessá-lo.

```
fclose(fp);
```

Capítulo 5

Vetores

Vetores são sequências de valores do mesmo tipo referenciados por um mesmo nome. Um elemento específico da sequência é acessado por meio de um índice inteiro não negativo.

Na linguagem C, os vetores começam com o índice zero. Um vetor com `n` posições terá como índices válidos os inteiros de zero a `n-1`.

5.1 Vetores unidimensionais

Uma declaração de vetor tem o seguinte formato:

```
<Tipo de dado> <Identificador> [ <Tamanho> ];
```

A expressão elementar abaixo funciona como uma variável comum, podendo ser usada em expressões aritméticas, de comparação, atribuição, ou como parâmetro em chamadas de funções.

```
<Identificador> [ <Posição> ]
```

Uma atribuição de valor a uma certa posição do vetor, por exemplo, tem o seguinte formato:

```
<Identificador> [ <Posição> ] = <Valor>;
```

5.2 Matrizes

Uma matriz é um vetor bidimensional. Sua declaração é semelhante à de um vetor unidimensional.

```
<Tipo de dado> <Identificador> [ <Linhas> ][ <Colunas> ];
```

Para acessar um item individual de uma matriz, usar o formato abaixo:

```
<Identificador> [ <Linha> ][ <Coluna> ]
```

Capítulo 6

Funções

Funções são blocos de instruções que executam uma certa tarefa. Podem receber dados de entrada e retornar dados de saída. Podemos dizer que a definição de uma função tem cinco partes: identificador ou nome, tipo de dado de retorno, lista de parâmetros, bloco de instruções e valor de retorno.

- Uma declaração de função tem a seguinte forma geral:

```
<Tipo de dado> <Identificador> (<Lista de parâmetros>)
{
      <Bloco de código>
      return <Valor de retorno>;
}
```

- Identificador é o nome através do qual chamamos a função. Alguns identificadores de funções já vistas são por exemplo `printf`, `scanf`, `fgets`, `sin` e `cos`.

- Tipo de dado de retorno é o tipo de dado que a função retorna. Como já foi visto na Seção 1.12, uma chamada a uma função é uma expressão elementar, e pode ser usada da mesma maneira que uma variável ou valor de mesmo tipo.

```
double x = sin(a);
printf("%f\n", x);
```

  Observe que `printf` imprimirá o valor de tipo `double` retornado pela função `sin`. De maneira alternativa, poderíamos colocar a chamada `sin` diretamente como parãmetro de `printf`.

```
printf("%f\n", sin(a));
```

- Lista de parâmetros são os dados de entrada da função. Cada parâmetro é uma variável cujo valor é definido na chamada da função. Cada item da lista tem o mesmo formato de uma declaração de variável, ou seja, é um tipo de dado seguido de um identificador. Se uma função recebe mais de um parâmetro, eles devem ser separados por vírgula.
- Bloco de código é o código executado quando a função é chamada.
- Valor de retorno é o valor retornado para a função que fez a chamada. Uma chamada a função pode fazer parte de uma expressão e tem o mesmo efeito que seu valor de retorno.

No exemplo abaixo podemos ver uma declaração bem simples de função. Essa função não recebe parâmetros e nem retorna valor. Simplesmente imprime uma mensagem e retorna. Logo após a declaração, vemos um exemplo de função `main` que a chama.

```
#include <stdio.h>
void cumprimenta(void)
{
  printf("Bom dia");
}

int main(void)
{
  cumprimenta();
}
```

Abaixo temos outro exemplo de função que não retorna valores. Essa função recebe um número de tipo `double` e o imprime usando notação científica.

```
#include <stdio.h>
void imprimed(double val)
{
  printf("%.10E", val);
}

int main(void)
{
  imprimed(299792458.0);
  printf("\n");
}
```

Vale notar que o tipo de dado de retorno da função não tem relação com os tipos dos parãmetros. Podem ser completamente diferentes como no exemplo abaixo, que recebe `float` como entrada e retorna `int`. A expressão `(int)` executa um *typecast*, ou seja, uma conversão de tipo.

```c
#include <stdio.h>
int arredonda(float val)
{
  return (int) (val+0.5);
}

int main(void)
{
   printf("Pi eh aproximadamente %d\n",
       arredonda(3.14159265));
}
```

6.1 Passagem de parâmetros por valor

Quando chamamos uma função, os valores dos parãmetros são copiados para as variáveis definidas na lista de parâmetros. Podemos modificar esses valores no interior da função sem que isso tenha efeito fora dela.

```c
#include <stdio.h>
int dobra(int n)
{
  n = 2 * n;
  return n;
}

int main(void)
{
  int a = 5;
  int b = dobro(a);
  printf("O dobro de %d eh %d\n", a, b);
}
```

O valor de `a` é copiado para o parâmetro `n` da função. `n` recebe 10 mas `a` permanece 5.

6.2 Passagem de parâmetros por referência

Esse tipo de passagem de parâmetro é necessário quando a função deve ser capaz de alterar alguma variável fora de seu escopo. Em outras palavras, quando precisamos que a função altere alguma variável da função que a chamou. Um exemplo de passagem de parâmetro por referência é o que fazemos ao chamar `scanf`.

Nesse tipo de passagem de parâmetro, ao invés de passarmos um valor, passamos um ponteiro, ou seja, uma referência à posição de memória que gostaríamos de alterar.

No exemplo abaixo, a função `troca` é implementada. Ela troca os valores de duas variáveis da função que a chamou. Isso só é possível usando passagem de parâmetro por referência.

```
#include <stdio.h>
int troca(int *a, int *b)
{
  int aux = *a;
  *a = *b;
  *b = aux;
}

int main(void)
{
  int a = 5;
  int b = 3
  printf("Antes: x = %d, y = %d\n", x, y);
  troca(&x, &y);
  printf("Depois: x = %d, y = %d\n", x, y);
}
```

Capítulo 7

Exercícios

7.1 Introdução

1- Crie uma função `main` vazia.

2- Crie uma função `main` e dentro dela declare as seguintes variáveis:
   a) um inteiro chamado `n`.
   b) um real de precisão simples chamado `f`.
   c) um real de precisão dupla chamado `d`.
   d) um caracter chamado `c`.
   e) um inteiro longo chamado `l`.

3- Em C, as instruções devem ser terminadas com ponto e vírgula. A declaração de uma variável é o primeiro lugar do programa onde a variável aparece e é onde seu tipo de dados é definido. A inicialização é onde algum valor é atribuído a ela pela primeira vez. A declaração e inicialização podem ocorrer na mesma instrução. Por exemplo:

```
int i = 0;
float x = 5.2;
float y = x*x;
```

Crie uma função `main` e dentro dela declare e inicialize na mesma instrução:
   a) um inteiro chamado `val`.
   b) um real de precisao simples chamado `x`.
   c) um real de precisão dupla chamado `xd`.
   d) um caracter chamado `tiny`.
   e) um inteiro longo chamado `huge`.

4- Crie uma função `main` que faz o mesmo que na questão anterior e em seguida,

para cada variável, imprime uma mensagem com seu nome, a palavra "vale", e seu valor. Para uma variável chamada pipoca, por exemplo:

```
pipoca vale 3.141592
```

5- Crie uma função `main` que faz o mesmo que na questão anterior, porém, ao invés de declarar variáveis, declare constantes.

6- Crie uma função `main` e dentro dela declare e inicialize na mesma instrução cinco variáveis inteiras. Para inicializá-las use expressões com três operadores aritméticos cada. Na mesma linha, coloque em um comentário o resultado da expressão. Por exemplo:

```
int i = 5 + 2 * 3 * 4; // 29
```

7- Crie uma função `main` e dentro dela declare e inicialize na mesma instrução cinco variáveis inteiras. Para inicializá-las use expressões com cinco operadores aritméticos cada. Na mesma linha, coloque em um comentário o resultado da expressão.

8- Crie uma função `main` e dentro dela declare e inicialize na mesma instrução cinco variáveis reais de precisão dupla. Para inicializá-las use expressões com dois operadores aritméticos e uma chamada a função em cada expressão. Na mesma linha, coloque em um comentário o resultado da expressão.

9- Crie uma função `main` e dentro dela declare e inicialize na mesma instrução cinco variáveis reais de precisão simples. Para inicializá-las use expressões com três operadores aritméticos cada. Na mesma linha, coloque em um comentário o resultado da expressão.

10- Crie uma função `main` que imprime a mensagem `"Hello\n"`.

11- Crie uma função `main` que imprime a mensagem `"Hello\n"` e em seguida chama `system("pause")`.

12- O que acontece quando:
a) atribuímos um valor `int` a uma variável `float`?
b) atribuímos um valor `float` a uma variável `int`?

7.2 Controle de fluxo

1- Crie uma função `main` que imprime todos os inteiros de 1 a 100 com apenas uma instrução com `printf`. Use:
a) um laço com a palavra reservada `while`.
b) um laço com a palavra reservada `for`.

2– Seja n um inteiro entre 1 e 1000. Crie uma função `main` com um laço (`for` ou `while`) que imprime uma mensagem `"%d modulo 2 vale %d"` para cada `n` entre 1 e 1000. No lugar do primeiro %d deve aparecer o valor de `n`, e no lugar do segundo %d, o valor de `n` modulo 2. O operador módulo, representado por %, retorna o resto da divisão de um número por outro número. Por exemplo, 12 % 10 retorna 2.

Resposta:

```
int main(void)
{
    for(int i = 1; i <= 1000; i++)
    {
        printf("\%d modulo 2 vale \%d\n", i, i \% 2);
    }
}
```

3– Seja n um inteiro entre 1 e 1000. Crie uma função `main` com um laço (for ou while) que imprime uma mensagem `"%d eh par"` se `n` for par, ou `"%d eh impar"` se `n` for impar, para cada n entre 1 e 1000. No lugar do primeiro `%d` deve aparecer o valor de `n`. Usar if para testar se o módulo de `n` por 2 vale 0, o que indica que `n` é par, ou 1, o que indica que `n` é ímpar. Observe que operador de comparação que testa igualdade em C é o `==`, diferente do operador de atribuição `=`.

4– Seja $v = \sum_{i=0}^{10} i$. Crie uma função `main` que calcula v usando:
a) um laço com a palavra reservada `while`.
b) um laço com a palavra reservada `for`.

5– Seja n um número inteiro. Crie uma função main que calcula seu fatorial usando um laço. Qual o maior n de tipo `int` que o C é capaz de calcular? E de tipo `long int`?

6– O número de combinações distintas de k elementos extraídos de um conjunto de n elementos é denotado por $C(n, k)$. Esse valor pode ser calculado pela fórmula $C(n, k) = n!/(k!(n - k)!)$. Seja $n = 8$. Crie uma função `main` que imprime em uma linha todos os valores de $C(n, k)$, com $0 \leq k \leq n$. Sugestão: use um `for` dentro de outro `for`.

7– Crie uma função `main` que faz o mesmo que o programa anterior para $n = 0$ até $n = 8$. A saída terá portanto 9 linhas.

8– Considere a equação de segundo grau $f = ax^2 + bx + c = 0$. Calcule uma de

suas raízes usando o método de Newton. Atribua às variáveis `a`, `b` e `c` os valores dos coeficientes da equação, atribua à variável `x0` um valor de chute inicial, e à variável `epsilon` uma tolerância. A cada passo encontre um novo chute inicial `x0` usando os resultados de $f(x0)$ e $f'(x0)$. O laço pára quando $f(x0)$ for menor que epsilon. Dica: $dx = x0 - x1$ e $f'(x0) = f(x0)/dx$.

7.3 Funções

1- Crie uma função chamada `alo` que não recebe parâmetros nem retorna valores, apenas imprime a mensagem `"hello world"`. Resposta:

```
void alo(void)
{
  printf("hello world\n");
}
```

2– Crie uma função `main` que simplesmente chama a função `alo`.

3- Crie uma função de tipo `void` chamada `preguica`, que não recebe parâmetros e não faz nada.
4– Crie uma função `main` que simplesmente chama a função `preguica`.

5- Crie uma função chamada `imprimeInt` que recebe um parâmetro inteiro, mas não retorna valores, apenas imprime a mensagem `"valor inteiro igual a "` seguida do valor do parâmetro recebido.

6– Crie uma função `main` que simplesmente chama a função `imprimeInt`. Lembre-se de fornecer como parâmetro algum valor inteiro.

7- Crie uma função chamada `duzia` que recebe um parâmetro real de precisão simples e retorna 12 vezes seu valor. Resposta:

```
float duzia(float x)
{
  return 12*x;
}
```

8– Crie uma função `main` que simplesmente chama a função `duzia`. Lembre-se de fornecer como parâmetro algum valor de tipo `float`. Salve o resultado em uma variável e imprima o resultado.

9- Crie uma função de tipo `int` chamada `triplo`, que recebe um parâmetro

inteiro e retorna três vezes seu valor.
10– Crie uma função `main` que chama a função `triplo`, salva o valor retornado em uma variável e em seguida o imprime.

11- Crie uma função de tipo `int` chamada `eh_par`, que recebe um parâmetro inteiro e retorna 1 (um) caso o inteiro seja par, e 0 (zero) caso contrário.

12- Crie uma função de tipo `void` chamada `imprime_float`, que recebe um parâmetro de tipo float e imprime a mensagem:
`"o valor da variavel real eh %f\n"`
substituindo o `%f` pelo valor do parâmetro.

13- Crie uma funcão de tipo `float` chamada `circunferencia`, que recebe um parâmetro de tipo `float` igual ao raio de um círculo e retorna o comprimento de sua circunferência.

14- Crie uma funcão de tipo `float` chamada `area_circulo`, que recebe um parâmetro de tipo `float` igual ao raio de um círculo e retorna seu raio.

15- Crie uma funcão de tipo `float` chamada `volume_esfera`, que recebe um parâmetro de tipo `float` igual ao raio de uma esfera e retorna seu volume.

16- Crie uma funcão de tipo `int` chamada `somatoria`, que recebe um parâmetro de tipo `int` chamado `n` e retorna a soma dos inteiros de 1 a `n`.

17- Crie uma função chamada `fatorial` que recebe um parâmetro inteiro e retorna seu fatorial.
a) implemente a função usando um laço while.
b) implemente a função usando um laço for.
c) implemente a função usando uma chamada a ela mesma. Para isso, use a definição recursiva de fatorial:
se `n == 0` (`==` é operador de comparação)
então `fatorial(n) = 1`
senão `fatorial(n) = n * fatorial(n-1)`

18- Crie uma funcão de tipo `int` chamada `somatoria`, que recebe um parâmetro de tipo `int` chamado `n` e retorna a soma dos inteiros de 1 a `n`.

19- Crie uma funcão de tipo `double` chamada `norma`, que recebe dois parâmetros de tipo `double` chamados `a` e `b`, e retorna a norma do vetor (a,b).

20- Crie uma função chamada `pg3`, que retorna o terceiro termo de uma progressão geométrica. A função recebe dois parâmetros inteiros, o primeiro parâmetro é o primeiro termo da pg; o segundo é sua razão.

21- Crie uma função chamada `pg`, que retorna o n-ésimo termo de uma progressão

geométrica. A função recebe três parâmetros inteiros, o primeiro parâmetro é o primeiro termo da pg; o segundo é sua razão, o terceiro é o `n`.

22-

a) Crie uma função de tipo `void` chamada `swap` que recebe dois parâmetros inteiros por referência e troca seus valores.

b) Crie uma função `main` e dentro dela declare e inicialize duas variáveis inteiras. Imprima seus valores, chame a função `swap` passando-as por referência e imprima novamente, para testar se os valores foram realmente trocados.

23- Crie uma funcão de tipo `double` chamada `delta`, que recebe três parâmetros de tipo `double` chamados `a`, `b` e `c`, e retorna o valor de delta, ou seja, b*b – 4.0*a*c.

24– Crie uma função de tipo `int` chamada `baskara`. Ela recebe por valor três parâmetros de tipo double chamados `a`, `b` e `c`, representando os coeficientes de uma equação de segundo grau igual a a*x*x + b*x + c. Recebe ainda dois parâmetros por referência, chamados `x1` e `x2`, onde armazenará as raízes encontradas. Ela retorna o número de raízes reais da equação, ou seja, pode retornar 0 (zero), 1 (um) ou 2 (dois). Usar a função delta da questão anterior para testar quantas raízes a equação possui.

7.4 Entrada e saída

1- Crie uma funcão `main` e dentro dela:

a) imprima a mensagem `"Favor digitar um valor inteiro \n"`.

b) declare uma variável inteira chamada `i`.

c) use `scanf` para ler o inteiro digitado pelo usuário e salvá-lo na variável `i`.

d) imprima a mensagem `"Voce digitou "` seguida do valor digitado e `"\n"`.

e) chame `system("pause")`.

2- Crie uma funcão `main` e dentro dela:

a) imprima a mensagem `"Favor digitar um valor inteiro\n"`.

b) declare as variáveis inteiras `i` e `result`.

c) use `scanf` para ler o inteiro digitado pelo usuário e salvá-lo na variável `i`. Armazene o valor de retorno de `scanf` em `result`.

d) caso o valor de `result` seja igual a 1 (um), imprima a mensagem `Voce digitou "` seguida do valor digitado e `"\n"`.

e) caso contrário, imprima a mensagem `"Eu pedi para digitar um inteiro!\n"`.

f) chame `system("pause")`.

3- Crie uma funcão `main` e dentro dela:

a) solicite ao usuário que forneça dois valores reais e armazene-os em duas variáveis chamadas `a` e `b`.

b) para cada variável lida, se a leitura foi bem sucedida imprima o valor lido. Caso contrário imprima uma mensagem de erro e finalize o programa.

c) imprima mensagens indicando o valor da soma, do produto, da diferença, da razão entre `a` e `b`, norma do vetor (a,b) e o valor de `a` elevado a `b`.

4- Crie uma funcão `main` e dentro dela:

a) solicite ao usuário que forneça três valores reais e armazene-os em três variáveis chamadas `a`, `b` e `c`.

b) para cada variável lida, se a leitura foi bem sucedida imprima o valor lido. Caso contrário imprima uma mensagem de erro e finalize o programa.

c) imprima mensagens indicando o valor das raízes da equação de segundo grau igual a a*x*x + b*x + c. Usar as funções das questões `baskara` e `delta` criadas anteriormente.

7.5 Vetores

1- Crie uma funcão `main`:

a) antes dela defina `TAM_MAX` como sendo 10 (dez).

b) dentro dela declare um vetor de inteiros chamado `vet`, com tamanho máximo igual `TAM_MAX`.

c) declare uma variável inteira chamada `tamanho`, para armazenar o tamanho ocupado desse vetor. Inicialize-a com zero.

d) crie um laço `for` que, a cada passo, solicita ao usuário que insira valores para cada posição do vetor, lê e armazena o dado digitado.

e) em caso de sucesso do `scanf`, incremente `tamanho` e continue. Em caso de erro, interrompa o laço com `break`.

f) depois do laço, imprima o tamanho ocupado do vetor. Note que o tamanho ocupado pode ser diferente do tamanho máximo `TAM_MAX`.

g) imprima o conteúdo de cada uma das posições do vetor usando a seguinte mensagem: `vet[%d] = %d\n"` substituindo o primeiro `%d` pela posição, e o segundo pelo conteúdo.

2- Crie uma funcão `main` e dentro dela, declare um vetor de reais chamado `numeros`. Seu tamanho máximo é definido por uma macro chamada `TAM_MAX`, que vale 10 (dez). Em seguida, abra um arquivo chamado `input.txt` cujo conteúdo é um valor real em cada linha. Use `fscanf` para ler esse arquivo e armazenar os valores no vetor `numeros`, e o tamanho ocupado do vetor na variável inteira chamada `tamanho`. Seu programa deve funcionar para qualquer tamanho de entrada, inclusive para arquivos com mais de dez valores. Nesse caso, o vetor ficará completamente ocupado e o valor de tamanho será 10.

Capítulo 8

Exemplos de programas

8.1 pause.cpp

```
/* Programa que imprime uma mensagem e termina quando o
   usuario pressiona enter.
*/

#include <stdio.h>

int main(void)
{
    printf("Pressione enter para continuar...");
    getc(stdin);
}
```

Listagem 8.1: series.cpp

8.2 ascii.cpp

```
/* Programa que imprime os 128 caracteres da tabela
   ASCII.
*/

# include <stdio.h>

#define NUMCOL  4
#define MAXCHAR 127

main()
{
    const char* arrName[] = {"NUL", "STX", "SOT", "ETX",
        "EOT", "ENQ", "ACK", "BEL", "BS", "HT", "LF",
```

```
        "VT", "FF", "CR", "SO", "SI", "DLE", "DC1",
        "DC2", "DC3", "DC4", "NAK", "SYN", "ETB", "CAN",
        "EM", "SUB", "ESC", "FS", "GS", "RS", "US"};
    const int arrNameSize = 32;

    for(int i = 0; i < (MAXCHAR + 1) / NUMCOL; i++)
    {
        for(int j = 0; j < NUMCOL; j++)
        {
            int c = j * ((MAXCHAR + 1) / NUMCOL) + i;
            if (c < arrNameSize)
            {
                printf("| %3d 0x%02X %3s ", c, c,
                    arrName[c]);
            }
            else
            {
                printf("| %3d 0x%02X %3c ", c, c, c);
            }
        }
        printf("|\n");
    }
}
```

Listagem 8.2: ascii.cpp

8.3 dado.cpp

```
/* Programa que lanca um dado com tantos lados quanto
   definidos em LADOS e imprime o valor sorteado.
   Cada sorteio resulta em um valor maior
   ou igual a 1 e menor ou igual a LADOS.
   O numero de sorteios eh definido em SORTEIOS.
   O programa conta quantas vezes cada
   lado foi sorteado e imprime a contagem no final.
*/



#include <stdio.h>  // printf, getchar
#include <stdlib.h> // rand, srand
#include <time.h>   // time

#define LADOS 6
#define SORTEIOS 10000
```

```
int main(void)
{

    srand(time(NULL));
    int count[LADOS + 1];

    for (int i = 0; i <= LADOS; i++)
    {
        count[i] = 0;
    }

    for (int i = 0; i < SORTEIOS; i++)
    {
        int result = rand() % LADOS + 1;
        printf("valor = %d\n", result % LADOS + 1);
        //getchar();
        count[result]++;
    }

    for (int i = 0; i <= LADOS; i++)
    {
        printf("count[%d] = %d\n", i, count[i]);
    }
}
```

Listagem 8.3: dado.cpp

8.4 primos.cpp

```
/*  Este programa solicita ao usuario que digite um
    numero inteiro. O numero eh lido e armazenado em
    uma variavel. Em seguida o programa calcula seus
    fatores primos e os imprime.
 */

#include <stdio.h>
#include <cstdlib>

int main()
{
    int val;
    printf("digite um inteiro:\n");
    int result = scanf("%d", &val);
    printf("result = %d, val = %d\n", result, val);

    int quociente = val;
```

```
    for (int i = 2; i <= val; i++)
    {
        int resto;
        do
        {
            resto = quociente % i;
            //printf("resto = %d, quoc = %d, i = %d\n",
                resto, quociente, i);
            if (resto == 0)
            {
                printf("fator: %d\n", i);
                quociente = quociente / i;
            }
        }
        while (resto == 0);
    }

    system("pause");
}
```

Listagem 8.4: series.cpp

8.5 series.cpp

```
/* Este programa solicita ao usuario que digite um
    numero inteiro. O numero eh lido e armazenado em
    uma variavel. Em seguida o programa calcula seu
    fatorial, seu namero de fibonacci e imprime o
    resultado.
 */

#include <stdio.h>
#include <cstdlib>

int fibonacci(int val)
{
    if (val == 0)
    {
        return 0;
    }
    else if (val == 1)
    {
        return 1;
    }
    else
    {
```

```
        return fibonacci(val - 1) + fibonacci(val - 2);
    }
}

int fatorial(int val)
{
    if (val <= 1)
    {
        return 1;
    }
    else
    {
        return val * fatorial(val - 1);
    }
}

int main()
{
    int val;
    printf("digite um inteiro:\n");

    int coderr = scanf("%d", &val);
    printf("coderr = %d, val = %d\n", coderr, val);

    int result;

    result = fatorial(val);
    printf("fatorial(%d) = %d\n", val, result);

    result = fibonacci(val);
    printf("fibonacci(%d) = %d\n", val, result);

    system("pause");
}
```

Listagem 8.5: series.cpp

8.6 generate.cpp

```
/* Gera uma lista de numeros aleatorios entre 0 e MAXNUM
   inclusive. Passar como parametro um inteiro igual a
   quantitade de numeros a serem gerados.
   Para criar um arquivo com os numeros gerados, suceder
   a chamada ao programa com > e o nome do arquivo.
   Exemplo:
   generate 1000 > saida.txt
```

```
*/

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

#define DEBUG  0
#define MAXNUM 100

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

  if (DEBUG){
    printf("argc = %d\n", argc);
    for (int i = 0; i < argc; i++)
    {
      printf("argv[%d] = %s\n", i, argv[i]);
    }
  }

  if (argc < 2)
  {
    exit(1);
  }

  int size;
  int result = sscanf(argv[1], "%d", &size);

  if (result != 1)
  {
    exit(1);
  }

  for (int i = 0; i < size; i++)
  {
    int number = rand() % MAXNUM + 1;
    printf("%d\n", number);
  }
}
```

Listagem 8.6: generate.cpp

8.7 ordena.cpp

```
/* Este programa recebe como parametro um nome
   de arquivo com um inteiro em cada linha.
   Le o arquivo, armazena os inteiros em um
```

```
    vetor e os ordena, imprimindo o tempo
    gasto na ordenacao.
 */

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

#define MAXTAM 1000000

#define swap(arr, a, b)          \
{                                \
    int __swap_buf__ = arr[a]; \
    arr[a] = arr[b];             \
    arr[b] = __swap_buf__;       \
}

int readArrayFromFile(char* filename, int* arr, int*
    size)
{
    FILE* fp = fopen(filename, "r");

    if (fp == 0)
    {
      return 1;
    }

    int result;
    int i = 0;
    do
    {
        result = fscanf(fp, "%d", &arr[i]);
        i++;
    }
    while (result == 1);
    *size = i - 1;

    return 0;
}

void printArray(int* arr, int size)
{
    for (int i = 0; i < size; i++)
    {
        printf("arr[%7d] = %7d\n", i, arr[i]);
    }
```

```
}

void bubbleSort(int* arr, int size)
{
    for (int maxi = size - 2; maxi >= 1; maxi--)
    {
        for (int i = 0; i <= maxi; i++)
        {
            if (arr[i] > arr[i+1])
            {
                swap(arr, i, i+1);
            }
        }
    }
}

double timer(int start)
{
    static timespec t0;
    static timespec t1;

    if (start)
    {
        clock_gettime(CLOCK_REALTIME, &t0);
        printf("t0 sec  = %ld\n", t0.tv_sec);
        printf("t0 nsec = %ld\n", t0.tv_nsec);
        return 0.0;
    }
    else
    {
        clock_gettime(CLOCK_REALTIME, &t1);
        printf("t1 sec  = %ld\n", t1.tv_sec);
        printf("t1 nsec = %ld\n", t1.tv_nsec);

        double delta =   (t1.tv_sec - t0.tv_sec) +
                       ( (t1.tv_nsec - t0.tv_nsec) /
                           1000000000.0 );

        return delta;
    }

}

void timerStart(void)
{
    timer(1);
```

```
}

double timerEnd(void)
{
  return timer(0);
}

int main(int argc, char** argv)
{
    if (argc < 2)
    {
        exit(1);
    }

    int arr[MAXTAM];

    int size;
    int error = readArrayFromFile(argv[1], arr, &size);

    if (error)
    {
      exit(1);
    }

    printf("size = %d\n", size);
    //printArray(arr, size);

    printf("CLOCKS_PER_SEC = %ld\n", CLOCKS_PER_SEC);

    timerStart();
    bubbleSort(arr, size);
    double delta = timerEnd();

    printf("delta = %f sec\n", delta);
    //printArray(arr, size);
}
```

Listagem 8.7: ordena.cpp

Apêndice A

Tabela ASCII

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0x00	NUL	32	0x20		64	0x40	@	96	0x60	‘
1	0x01	STX	33	0x21	!	65	0x41	A	97	0x61	a
2	0x02	SOT	34	0x22	"	66	0x42	B	98	0x62	b
3	0x03	ETX	35	0x23	#	67	0x43	C	99	0x63	c
4	0x04	EOT	36	0x24	$	68	0x44	D	100	0x64	d
5	0x05	ENQ	37	0x25	%	69	0x45	E	101	0x65	e
6	0x06	ACK	38	0x26	&	70	0x46	F	102	0x66	f
7	0x07	BEL	39	0x27	’	71	0x47	G	103	0x67	g
8	0x08	BS	40	0x28	(	72	0x48	H	104	0x68	h
9	0x09	HT	41	0x29	)	73	0x49	I	105	0x69	i
10	0x0A	LF	42	0x2A	*	74	0x4A	J	106	0x6A	j
11	0x0B	VT	43	0x2B	+	75	0x4B	K	107	0x6B	k
12	0x0C	FF	44	0x2C	,	76	0x4C	L	108	0x6C	l
13	0x0D	CR	45	0x2D	-	77	0x4D	M	109	0x6D	m
14	0x0E	SO	46	0x2E	.	78	0x4E	N	110	0x6E	n
15	0x0F	SI	47	0x2F	/	79	0x4F	O	111	0x6F	o
16	0x10	DLE	48	0x30	0	80	0x50	P	112	0x70	p
17	0x11	DC1	49	0x31	1	81	0x51	Q	113	0x71	q
18	0x12	DC2	50	0x32	2	82	0x52	R	114	0x72	r
19	0x13	DC3	51	0x33	3	83	0x53	S	115	0x73	s
20	0x14	DC4	52	0x34	4	84	0x54	T	116	0x74	t
21	0x15	NAK	53	0x35	5	85	0x55	U	117	0x75	u
22	0x16	SYN	54	0x36	6	86	0x56	V	118	0x76	v
23	0x17	ETB	55	0x37	7	87	0x57	W	119	0x77	w
24	0x18	CAN	56	0x38	8	88	0x58	X	120	0x78	x
25	0x19	EM	57	0x39	9	89	0x59	Y	121	0x79	y
26	0x1A	SUB	58	0x3A	:	90	0x5A	Z	122	0x7A	z
27	0x1B	ESC	59	0x3B	;	91	0x5B	[	123	0x7B	{
28	0x1C	FS	60	0x3C	<	92	0x5C		124	0x7C	|
29	0x1D	GS	61	0x3D	=	93	0x5D	]	125	0x7D	}
30	0x1E	RS	62	0x3E	>	94	0x5E	^	126	0x7E	
31	0x1F	US	63	0x3F	?	95	0x5F	_	127	0x7F	DEL