Sumário

Capítulo 1

Histórico

A linguagem `C` foi criada por Dennis M. Ritchie e Ken Thompson nos laboratórios Bell em 1972, baseada na linguagem B de Thompson que era uma evolução da antiga linguagem BCPL. Ela foi desenvolvida para o sistema operacional UNIX, que por sua vez foi totalmente desenvolvido em `C` .

A linguagem `C` (de agora em diante somente `C`) tornou-se vitoriosa como ferramenta de programação para qualquer tipo de sistema (sistemas operacionais, planilhas eletrônicas etc.) pela sua portabilidade (capacidade de um programa poder rodar em qualquer plataforma, só sendo preciso recompilá-lo), flexibilidade e por seu poder de expressão, além da padronização dos compiladores existentes (um adendo à portabilidade). `C` foi desenvolvida para que o usuário pudessea planejar programas estruturados e modulares, resultando em programas mais legíveis e documentados. Os programas em `C` tendem a ser bastante compactos e de rápida execução.

`C` é uma linguagem amigável e suficientemente estruturada para encorajar bons hábitos de programação, possibilitando também o desenvolvimento de partes separadas de um programa por pessoas distintas. Essas partes podem ser reunidas finalmente em um produto final, o que significa que bibliotecas de funções podem ser criadas ou usadas sem realmente se conhecer o código de cada uma delas.

Existem muitas outras virtudes de C que você conhecerá ao longo de seu aprendizado. A referência básica para o conhecimento de `C` é [RIT86].

Capítulo 2

Conceitos básicos

2.1 Estrutura básica de um programa em C

Vamos começar com um programa bem simples, mas que contém muitas das características de um programa C , suficientes para começarmos nossa discussão.

—Exemplo 1: *hello.c* –

```
main()
{
        printf("Alo, mundo!!!\n");
}
```

O primeiro elemento que se pode observar é a função main(). Como o nome dela já diz, ela é a função principal de todo programa C . É ela que recebe os (possíveis) parâmetros passados em linha de comando e é a que retorna o controle para o processo (ex. o *shell*) que o executou.

O segundo aspecto é a função printf(). Mais uma vez, o nome explica sua funcionalidade: esta função serve para imprimir texto na tela (*print* é imprimir em inglês). O *f* é de *formatado*: os dados são impressos na tela de acordo com um formato especificado pelo usuário. Mais adiante veremos mais detalhes a respeito dessa função (vide Capítulo 4).

Note que logo após a chamada da função printf vem um '**;**', que é o caracter separador de comandos. Ao fim de cada comando deve-se colocar este caracter. Outro fator a ser observado são os caracteres "{" e "}", delimitando o *escopo* [1] da função main.

Em um segundo exemplo, temos um programa mais elaborado. Nele podemos observar uma estrutura um pouco mais complexa (sendo porém apenas uma extensão do exemplo acima).

[1] *Escopo* é o conjunto de definições de uma função. É a sua "área reservada".

–Exemplo 2: *Um exemplo mais elaborado* –

```
/* inclusao das bibliotecas */
#include<stdio.h>
#include<stdlib.h>

/* definicao de constantes */
#define PI 3.14

/* declaracao de funcoes (prototipos) */
float seno(int angulo);

/* declaracao de variaveis globais */
float raio;

/* corpo de comandos principal */
int main()
{
    /* comandos */
}

/* definicao (implementacao) de funcoes */
float seno(int angulo)
{
        /* uma variável local */
        int a;

    /* corpo da funcao */
}
```

Neste exemplo vemos várias

2.2 Variáveis

Uma variável é uma localização para armazenagem de dados na memória do computador. Quando o nome de uma variável aparece em um programa ele está se referindo, na verdade, aos dados armazenados nessa localização.

2.2.1 Nomes de Variáveis

Para usar variáveis nos programas em C, devemos saber quais nomes podem ser utilizados. Para tanto, devemos seguir as seguintes regras:

- O nome pode conter letras, algarismos e o caractere '_';
- O primeiro caractere do nome sempre deve ser uma letra ou o caractere '_';
- Letras maiúsculas são diferentes de minúsculas;
- As palavras-chave da linguagem C não podem ser utilizadas como nomes das variáveis;
- A escolha de nomes significativos para suas variáveis pode tornar o programa mais claro e fácil de entender.

Exemplos:

```
percent      /* válido */
y2x5_fg7h    /* válido */
double       /* inválido: é uma palavra chave */
9winter      /* inválido: inicia com algarismo */
```

2.2.2 Tipos de Variáveis

As variáveis da linguagem C enquadram-se em uma das seguintes categorias:

- Variáveis inteiras, que armazenam valores não-fracionais (ou seja, somente valores inteiros). Há dois tipos de variáveis inteiras:
  - variáveis inteiras com sinal, que podem conter valores positivos ou negativos;
  - variáveis inteiras sem sinal, que só podem assumir valores positivos.

  Operações matemáticas entre variáveis inteiras são, normalmente, muito rápidas.
- Variáveis de Ponto Flutuante, que contém valores com uma parte fracional (ou seja, números reais). No geral, as operações matemáticas são mais lentas para este tipo de variável.

Tipo de Variável	Palavra Chave	Bytes	Valores Válidos
Caracteres	char	1	-128 a 127
Números inteiros curtos	short	2	-32.768 a 32.767
Números inteiros	int	4	-2.147.483.648 a 2.147.483.647
Caracteres não sinalizados	unsigned char	1	0 a 255
Números inteiros curtos sem sinal	unsigned short	2	0 a 65.535
Números inteiros sem sinal	unsigned long	4	0 a 4.294.967.295
Número de ponto flutuante com precisão simples	float	4	1,2E-38 a 3,4E38
Número de ponto flutuante com precisão dupla	double	8	2,2E-308 a 1,8E308

Tabela 2.1: Tipos de dados numéricos em C.

2.2.3 Declaração/Inicialização

Antes de poder ser usada por um programa, uma variável deve ser declarada. A declaração de uma variável informa ao compilador o nome e o tipo de uma variável e, opcionalmente, inicializa a variável com um determinado valor.

A declaração deve ter o seguinte formato:

```
tipo NomeVariavel;
```

Exemplo:

```
/* tres variaveis inteiras */
int count, number, start;

/* variavel de ponto flutuante inicializada com um valor */
float percent=15.3;
```

De acordo com o local no programa onde as variáveis são declaradas, elas se classificam em variáveis globais, se declaradas fora de qualquer função, e variáveis locais, se declaradas dentro de uma função. As variáveis globais são válidas em qualquer parte do programa e as variáveis locais apenas dentro do bloco em que foram declaradas.

2.3 Constantes

Uma constante, da mesma forma que uma variável, é uma localização usada pelo programa para armazenar dados. Ao contrário da variável, porém, o valor armazenado em uma constante não pode ser alterado durante a execução do programa. A linguagem C possui dois tipos de constantes: literais e simbólicas.

2.3.1 Literais

Constantes literais são os valores digitados diretamente no código-fonte do programa. Segue alguns exemplos de constantes literais:

```
int count = 20;
float tax_rate = 0.28;
char letter = 'c';
```

OBS.: Neste caso, os valores 20, 0.28 e 'c' são as constantes literais. Note que para especificar constantes que são caracteres, deve-se delimitar o valor com aspas simples.

2.3.2 Simbólicas

Constantes simbólicas são constantes representadas por um nome (símbolo) no programa.

Para utilizar o valor da constante no programa, podemos usar seu nome, exatamente como usaríamos o nome de uma variável. O nome da constante simbólica deve ser especificado somente uma vez, quando ela é definida. Por exemplo:

```
#define PI 3.14159
#define TAM_MAX 30

/* inclusão de bibliotecas */
#include <stdio.h>
#define PI 3.14159

void main()
{
  float perimetro;
  int raio = 5; /* atribui valor 5 ao raio */
  perimetro = PI*(2*raio); /* calcula o perimetro */
  printf("O perimetro da circunferencia de raio %d e %f",
      raio,perimetro);
}
```

A saída será: **O perimetro da circunferencia de raio 5 é 31.4159.**

2.3.3 Incluindo bibliotecas

Uma biblioteca é uma coleção de declarações de funções e constantes, que pode ser incluída em um programa e que provê uma série de "comandos" novos.

Para incluir uma biblioteca é utilizado a diretiva [2] *include*.

As sintaxes possíveis são:

- Inclusão de uma biblioteca residente em um diretório padrão:

  `#include <`*nomearq*`>`

- Inclusão de uma biblioteca residente no diretório local:

  `#include "`*nomearq*`"`

As diretivas acima incluem [3] a biblioteca *nomearq* no programa.

[2] Diretivas de pré-processamento serão melhor detalhadas no capítulo **??**.

[3] Na realidade, normalmente é realizada a inclusão de um *header*(cabeçalho) de uma biblioteca. Posteriormente este tópico será esclarecido.

2.4 Exercício

1. Para cada variável abaixo, determine a forma equivalente para declarar a mesma variável em C.

   a) $p \in N$
   b) $q \in Q$
   c) $r \in R$
   d) $s \in Z$
   e) sexo ([M]asc/[F]em)
   f) tamanho_camisa (P/M/G)

2. Qual tipo deveria ser utilizado para uma variável booleana[4]?

3. Qual a diferença de uma constante declarada através de *#DEFINE* e uma constante declarada utilizando a palavra-chave *const*?

4. Faça um programa completo em C que declara todas as variáveis do exercício 1.

[4]Variável que assume Verdadeiro ou Falso

Capítulo 3

Operadores

Um operador é um símbolo que faz com que uma determinada operação, ou ação, seja executada com um ou mais operandos. Um operando é aquilo sobre o qual o operador atua. Os operadores da linguagem C enquadram-se em diversas categorias.

3.1 Operador de Atribuição

O operador de atribuição é o sinal '='. Seu uso em programação é ligeiramente diferente de seu uso na Matemática normal. Se escrevermos

```
x = y;
```

em um programa em C, isto significa que o valor de y deve ser atribuído em x, e não que y é igual a x como seria de se esperar. Em uma instrução de atribuição, o lado direito pode ser qualquer expressão e o lado esquerdo deve necessariamente ser o nome de uma variável. A sintaxe, portanto, é:

```
variavel = expressao;
```

O operador de atribuição também pode ser usado de forma encadeada:

```
bananas = tomate = laranja = 50;
/* atribui o valor 50 a todas estas variaveis */
```

3.2 Operadores Matemáticos

Os operadores matemáticos são usados para realizar operações matemáticas, como adição ou subtração. A linguagem C possui dois operadores matemáticos unários e cinco binários.

3.2.1 Unários

Os operadores matemáticos unários recebem este nome porque exigem apenas um operando, e são os seguintes:

Operador	**Símbolo**	**Ação**	**Exemplo**
Incremento	++	Incrementa o operando em uma unidade	++x, x++
Decremento	−−	Decrementa o operando em uma unidade	−−x, x−−

Exemplos:

```
++x; /* equivalente a x = x + 1 */
y--; /* equivalente a y = y - 1 */
```

Observe que ambos os operadores podem ser colocados antes do operando *(modo de prefixo)* ou depois do operando *(modo de sufixo)*. Estes dois modos não são equivalentes.

A diferença está no momento em que o incremento ou decremento acontece.

- No modo de prefixo, os operadores de incremento e decremento modificam seu operando antes que este seja usado.
- No modo de sufixo, os operadores de incremento e decremento modificam seu operando depois que este é usado.

Exemplo:

```
x = 10;
y = x++;
```

Depois que ambas as instruções forem executadas, x terá o valor 11 e y o valor 10. O valor de x foi atribuído a y e em seguida o valor de x foi incrementado. Por outro lado, as instruções

```
x = 10;
y = ++x;
```

resultam no valor 11 tanto para x como para y, pois x é incrementado e só depois seu valor é atribuído a y.

```
#include <stdio.h>

void main()
{
  int a,b;

  a = b = 3; /* atribui o valor 3 a "a"e "b"*/
  /* Imprime ambas as variaveis e as decrementa usando*/
  /* o modo de prefixo para b e de sufixo para a */

  printf("\n%d %d",a--,--b);
  printf("\n%d %d",a--,--b);
  printf("\n%d %d",a--,--b);
}
```

A saída será:

3 2
2 1
1 0

3.2.2 Binários

Operadores binários são aqueles que exigem dois operandos. Os operadores binários da linguagem C, que incluem as operações matemáticas mais comuns, são relacionados na tabela 3.1

Operador	**Símbolo**	**Ação**	**Exemplo**
Adição	+	Soma seus dois operandos	x + y
Subtração	−	Subtrai o segundo operando do primeiro	x − y
Multiplicação	*	Multiplica seus dois operandos	x * y
Divisão	/	Divide o primeiro operando pelo segundo	x / y
Módulo	%	Fornece o resto da divisão do primeiro operando pelo segundo	x % y

Tabela 3.1: Operadores binários

Os primeiros quatro operadores já são familiares e não apresentarão qualquer problema. O operador módulo talvez seja novo. O módulo retorna o resto de uma divisão do primeiro operando pelo segundo.

Exemplo:

100 % 9 é igual a 1

10 % 5 é igual a 0

3.3 Atribuição Composta

Em C, os operadores de atribuição compostos permitem combinar uma operação matemática binária com uma operação de atribuição de valor. Em geral, estes operadores usam a seguinte sintaxe (sendo *op* um operador binário):

```
exp1 op= exp2;
```

que é equivalente a escrever

```
exp1 = exp1 op exp2;
```

Os operadores compostos existem para os cinco operadores matemáticos binários descritos anteriormente, conforme ilustra a tabela 3.2.

Se você escrever	**será equivalente a**
x += y	x = x + y
x − = y + 1	x = x − (y + 1)
x *= y	x = x * y
x /= y − 3	x = x / (y−3)
x %= y	x = x % y

Tabela 3.2: Operadores de atribuição compostos

3.4 Operadores Relacionais

São usados para comparar expressões. Uma expressão que contenha um operador relacional sempre é avaliada como verdadeira (1) ou falsa (0). Os seis operadores relacionais da linguagem C são listados na tabela 3.3.

Operador	**Símbolo**	**Pergunta Respondida**	**Exemplo**
Igual	==	Operando 1 é igual ao operando 2?	x == y
Maior que	>	Operando 1 é maior que o operando 2?	x > y
Menor que	<	Operando 1 é menor que o operando 2?	x < y
Maior ou igual a	>=	Operando 1 é maior ou igual ao operando 2?	x >= y
Menor ou igual a	<=	Operando 1 é menor ou igual ao operando 2?	x <= y
Diferente	!=	Operando 1 é diferente do operando 2?	x != y

Tabela 3.3: Operadores relacionais

3.5 Operadores Lógicos

Em certas situações, precisamos fazer mais de um teste relacional ao mesmo tempo. Os operadores lógicos permitem que combinemos duas ou mais expressões relacionais em uma única expressão, que é avaliada como verdadeira ou falsa.

Os três operadores lógicos da linguagem C são listados na tabela 3.4.

Operador	**Símbolo**	**Verdadeiro Quando**	**Exemplo**
And	&&	Expressão 1 **E** expressão 2 são verdadeiras	exp1 && exp2
Or	||	Expressão 1 **OU** expressão 2 são verdadeiras	exp1 || exp2
Not	!	A expressão é falsa	!exp1

Tabela 3.4: Operadores lógicos

3.6 Operador Condicional

O operador condicional é o único operador ternário (que exige três operandos) na linguagem C. Sua sintaxe é:

```
exp1 ?  exp2 :  exp3
```

Se a expressão exp1 for verdadeira (ou seja, não-zero), toda a expressão será avaliada como o valor de exp2. Se exp1 for falsa (ou seja, zero), toda a expressão será avaliada como o valor de exp3.

Exemplo:

```
z = (x > y) ?  x :  y;
/* Se x for maior que y, z=x, caso contrario, z=y */
```

Assim como na matemática, existe diferença na precedência dos operadores em C.

A tabela 3.5 lista todos os operadores e suas precedências. Sempre que uma expressão é encontrada, os operadores que têm maior precedência são avaliados antes. Se houver empate, ou seja, dois operandos com a mesma precedência, será avaliado antes o operador que estiver mais à esquerda.

Operadores	**Precedência**
! ++ −−	1
* & / %	2
+ − |	3
<<=>>=	4
== !=	5
&&	6
||	7
?:	8
= += − = *= /= %=	9

Tabela 3.5: Precedência de operadores

3.7 Exercícios

1. Escreva as fórmulas e expressões abaixo em linguagem C. Utilize apenas operadores e funções (quando aplicável).

   a) $a^2 + \frac{b}{c} \times \sqrt{2}$

   b) $(A \oplus B) \wedge C$

   c) C = 400 se A>B

   d) (A>B ou A=B) e C=30

   e) $\frac{a^2+b^2c^3}{\sqrt{\frac{q}{p}} \times \frac{r \times t}{m}} \times (\sin^2 x - \cos^2 x)$

2. Escreva um programa completo em C para calcular cada expressão acima. O programa deve incluir todas as bibliotecas necessárias e cada variável deve ser declarada apropriadamente.

3. Escreva um programa que recebe uma quantidade de tempo em segundos e converte-o para a forma horas : minutos : segundos.

   Não se preocupe com entrada de dados. Suponha que o tempo já está armazenado em uma variável t, declare as variáveis necessárias e realize as operações convenientes para calcular cada parâmetro.

   Dica: crie uma variável para cada parâmetro (h/m/s) e utilize o operador % para realizar os cálculos.

Capítulo 4

Funções de entrada e saída

Funções de entrada e saída são aquelas que obtêm informações de uma determinada entrada (normalmente através da *entrada padrão, i.e. teclado*) e enviam informações para uma determinada saída (normalmente para a *saída padrão, i.e. vídeo*).

Nesta seção estaremos tratando apenas de funções que trabalham com entrada e saída padrão, ou seja, obtêm uma informação a partir do teclado e imprimem informações para o vídeo. A biblioteca que contêm todas essas funções é a **stdio.h**.

4.1 printf()

A função *printf()* é a maneira mais fácil de fazer com que um programa exiba informações na tela. Ela recebe dois argumentos: uma *string de formato*(obrigatória) e uma lista de argumentos (opcional).

Uma *string de formato* especifica como a saída da função *printf* deverá ser formatada. Os três componentes possíveis de uma *string de formato* são:

- Texto literal, que é exibido na tela exatamente como foi incluído na string de formato.
- Seqüências de escape, que incluem instruções especiais de formatação. Uma seqüência de escape consiste de uma barra invertida (\) seguida de um único caractere. As seqüências de escape mais utilizadas estão na tabela 4.1.

Seqüência	**Significado**
\a	Sinal Sonoro
\b	retrocesso (backspace)
\n	nova linha
\t	tabulação horizontal
\\	barra invertida
\?	ponto de interrogação
\’	aspa simples
\”	aspas duplas

Tabela 4.1: Sequências de escape

- Especificadores de formatação, que consistem no símbolo de porcentagem (%) seguido por um único caractere. Estes especificadores informam à função printf() como interpretar as variáveis que serão impressas. O string de formato deve conter um especificador de formatação para cada variável a ser impressa.

Especificador	Significado
%c	caractere simples
%d	número inteiro decimal com sinal
%u	número inteiro decimal sem sinal
%s	string alfanumérica
%f	número decimal com ponto flutuante
%e	número em notação científica

Tabela 4.2: Especificadores de formatação

```
#include <stdio.h>

void main()
{
  int x = -10;
  unsigned int y = 20;
  float z = 7.32;
  /* Exibe na tela o caractere nova linha e o texto */
  printf("\nTeste de printf");
  /* Exibe na tela texto, caractere nova linha */
  /* e valor de variaveis */
  printf("\nInteiro:  %d \nSem Sinal:  %u", x, y);
  printf("\nPonto Flutuante:  %f", z);
  printf("\nString Alfanumerica:  %s","Teste de Printf");
}
```

A saída será:

Teste de Printf
Inteiro: -10
Sem Sinal: 20
Ponto Flutuante: 7.32
String Alfanumerica: Teste de Printf

4.2 scanf()

A função *scanf()* lê dados do teclado de acordo com um formato especificado e atribui os dados recebidos a uma ou mais variáveis do programa. Assim como o *printf(), scanf()* também usa uma string de formato para descrever como os dados recebidos serão formatados. A string de formato utiliza os mesmos especificadores de formatação utilizados pela função *printf()*.

Além da string de formato, esta função recebe uma lista de argumentos, que devem ser passados por *referência* (precedidos do caractere &).

```
#include <stdio.h>
void main()
{
  float y;
  int x;

  printf("Digite um numero de ponto flutuante, depois um inteiro:  ");
  scanf("%f %d", &y, &x); /* Le os numeros do teclado */
  printf("\nVoce digitou %f e %d",y,x); /* imprime-os */
}
```

A saída será:

Digite um numero de ponto flutuante, depois um inteiro: 17.65 9
Voce digitou 17.65 e 9

4.3 getchar() e putchar()

A função *getchar()*, que está definida na biblioteca padrão *stdio.h*, obtém o próximo caractere da entrada padrão e o ecoa na tela. Esta função não aceita argumentos, e retorna o caractere lido.

A função *putchar()*, também definida na biblioteca padrão *stdio.h*, envia um caractere para a saída padrão. A função retorna o caractere que acabou de ser enviado ou EOF em caso de erro.

```
#include <stdio.h>
void main()
{
  int ch;
  while((ch = getchar()) != '\n')
  putchar(ch);
}
```

Quando este programa for executado, a função *getchar()* é chamada e aguarda até receber um caractere da entrada padrão. Como *getchar()* é uma função de entrada com "buffer", nenhum caractere é recebido até que o usuário pressione a tecla **[enter]**. Não obstante, todas as teclas pressionadas são refletidas na tela.

Ao pressionar [enter], todos os caracteres digitados, inclusive o caractere nova linha, são enviados para a entrada padrão pelo sistema operacional. A função getchar() retorna os caracteres individualmente.

No programa acima, os caracteres recebidos pela função *getchar()* são armazenados na variável ch, que é então ecoada na tela utilizando-se a função *putchar()*.

4.4 Exercícios

4.4.1 Convenções

Antes de iniciar os exercícios deste capítulo, vamos estabelecer algumas convenções com relação ao enunciado do exercício.

Muito exercícios apresentarão um quadro da seguinte forma:

nonononononon <*varout*> nonononononononon
noononononono: >*varin*<
nonononononon <*(varopt:[n_1]*text*$_1$/[n_2]*text*$_2$/...)*>

Normalmente será necessário fazer um programa que imprime o texto contido no quadro. Duas notações especiais estão sendo usadas:

- <*varout*> : indica que o conteúdo da variável *varout* deve ser impresso no local indicado.

  Se entre <> houver palavras separadas por espaço (ex.: <a b c>), deverão ser impressos o conteúdo das variáveis *a*, *b* e *c*.

- <*(varopt:[n_1]*text*$_1$/[n_2]*text*$_2$/...)*>: indica que, de acordo com o valor de *varopt*, será impresso text$_n$ se varopt=n$_n$[1].

- >*varin*< : indica que o programa deverá requisitar ao usuário que digite um valor, o qual será armazenado em *varin*.

  Se entre >< houver palavras separadas por espaço (ex.: >a b c<), deverão ser lidos conteúdos para as variáveis *a*, *b* e *c*.

4.4.2 Exercícios

1. Complete os execícios do capítulo 3, incluindo funções de entrada para obter os valores a serem processados (onde for aplicável) e funções de saída para imprimir os resultados.

2. Determine os erros do programa abaixo e justifique suas conclusões.

[1] Em capítulos posteriores serão apresentados meios de tomar decisões baseadas no valor de uma expressão

```
#include <stdio.h>

#define PI 3.14159;

void main()
{
  float perimetro;
  int raio = /* atribuicao ao raio */ 5;
  const char letra_padrao = 65;
  const double d_var;
  char outra_letra;
  perimetro = PI*(2*raio);
  raio = 68.5;
  letra_padrao = raio;
  printf("A letra representada é %c.\n", letra_padrao);
  printf("Digite uma letra qualquer:");
  scanf("%d", outra_letra);
  printf("A letra digitada foi %c.\n", outra_letra);
  outra_letra = letra_padrao;
  }
```

3. Escreva um programa que seja capaz de imprimir os seguintes textos:

   a)
   Entre com um número: >num<
   O objetivo deste programa é apenas ilutrar o uso do printf e do scanf.
   O número que você digitou é <num>.
   Digite um número real: >numR<
   O número é <numR>, pertencente ao conjunto dos números reais.

   b)
   Escreva uma palavra que tenha exatamente 6 caracteres:
   :: >c0 c1 c2 c3 c4 c5<
   A palavra que você escreveu é: <c0 c1 c2 c3 c4 c5>
   A palavra revertida é: <c5 c4 c3 c2 c1 c0>

Capítulo 5

Estrutura de condição e loops

5.1 Formatos

A instrução *if* avalia uma expressão e conduz a execução do programa dependendo do resultado obtido. Ela utiliza os seguintes formatos:

- **Formato 1:**

```
if (expressão)
  instrução1;
próxima instrução;
```

  Quando a expressão é verdadeira, a instrução1 é executada. Quando a expressão é falsa, a instrução1 não é executada. Em ambos os casos, a próxima instrução é executada.

- **Formato 2:**

```
if (expressão)
  instrução1;
else
  instrução2;
próxima instrução;
```

  Quando a expressão é verdadeira, a instrução1 é executada e a instrução2 não. Quando a expressão é falsa, a instrução1 não é executada e a instrução2 é. Em ambos os casos, a próxima instrução é executada.

- **Formato 3:**

```
if (expressão1)
  instrução1;
else
  if (expressão2)
    instrução2;
  else
    instrução3;
  próxima instrução;
```

  Este formato utiliza uma instrução if aninhada. Se a expressão1 for verdadeira, a instrução1 será executada; caso contrário, a expressão2 será avaliada. Se esta for verdadeira, a instrução2 será executada, caso contrário, a instrução3 será executada. Em todos os casos, a próxima instrução é executada.

Uma instrução if pode controlar a execução de múltiplas instruções através de uma instrução composta, ou bloco. Um bloco é um conjunto de uma ou mais instruções delimitadas por chaves. Por exemplo:

```
if (expressao)
{
  instrucao1;
  instrucao2;
    ...
  instrucaoN;
}
```

```
/* Demonstra o uso da instrucao if */

#include <stdio.h>

void main()
{
  int x = 10, y = 7;

  if (x > y)
  {
    printf("x e maior que y\n");
    printf("O valor de x e %d", x);
  }
  else
    if (x < y)
    {
      printf("y e maior que x\n");
      printf("O valor de y e %d",y);
    }
  else
    printf("x e igual a y");
  }
```

A saída será:
x é maior que y
O valor de x é 10

5.2 Exercício

Faça um programa em C que, dadas as três notas e o número de matrícula de dois alunos, calcule e imprima a média e o estado de cada aluno conforme a tabela abaixo:

Média	**Estado**
< 40	Reprovado
< 70	Exame Final
¿= 70	Aprovado

Feito isto, informe o número de matrícula do aluno com maior e menor média, respectivamente.

5.3 Instrução *for*

A instrução *for* é um "loop"que permite repetir um determinado número de vezes a execução de um bloco contendo uma ou mais instruções. Ela tem a seguinte estrutura:

```
for(inicial; condição; incremento)
{
  instruções;
}
```

Neste caso inicial, condição e incremento são expressões válidas em C. Quando uma instrução *for* é encontrada durante a execução de um programa, os seguintes eventos ocorrem:

1. A expressão *inicial* é avaliada. Em geral, *inicial* é uma instrução de atribuição que inicializa uma variável com um determinado valor.
2. A expressão *condição* é avaliada. Tipicamente, *condição* é uma expressão relacional.
3. Se *condição* for falsa (ou seja, igual a zero), a instrução *for* termina a execução e passa para a instrução seguinte.
4. Se *condição* for verdadeira (ou seja, não zero), as instruções do *for* são executadas.
5. A expressão *incremento* é avaliada e a execução volta à etapa 2.

```
/* Exemplo que demonstra o uso da instrucao for */

#include <stdio.h>

void main ()
{
  int contagem;

  /* Imprime na tela os numeros de 1 a 10 */

  for(contagem = 1; contagem <= 10; contagem++)
    printf("%d",contagem);
}
```

A saída será: **1 2 3 4 5 6 7 8 9 10**

5.4 Instrução *while*

A instrução *while* é um "loop"que executa um bloco de instruções enquanto determinada condição permanecer verdadeira. A instrução *while* utiliza o seguinte formato:

```
while( condição )
{
  instruções;
}
```

A condição é qualquer expressão válida em C. Quando a execução do programa chega a uma instrução *while*, os seguintes eventos ocorrem:

1. A expressão *condição* é avaliada.
2. Se *condição* for falsa (ou seja, zero), a instrução *while* é terminada e a execução passa para a primeira instrução subseqüente.
3. Se *condição* for verdadeira (ou seja, diferente de zero), as instruções do *while* são executadas e a execução volta à etapa 1.

```
/* Demonstra o uso da instrucao while */
#include <stdio.h>

void main ()
{
```

```
  int contagem = 1;
  /* Imprime na tela os numeros de 1 a 10 */
  while ( contagem <= 10 )
  {
    printf("%d", contagem);
    contagem++;
  }
}
```

A saída será: **1 2 3 4 5 6 7 8 9 10**

5.5 Instrução *do...while*

A terceira estrutura de "loop" em C é o "loop" *do...while*, que executa um bloco de instruções enquanto uma determinada condição for verdadeira. Ao contrário do que fazem os "loops" *for* e *while*, o "loop" *do...while* testa a condição no final do "loop" e não no início. Ele usa a seguinte estrutura:

```
do {
  instruções;
}while (condição);
```

No caso, condição é qualquer instrução válida em C. Quando a execução do programa atinge uma instrução do...while, os seguintes eventos ocorrem:

1. São executadas as instruções do "loop".
2. *Condição* é avaliada. Se for verdadeira, a execução volta ao passo 1, senão, o "loop" é encerrado.

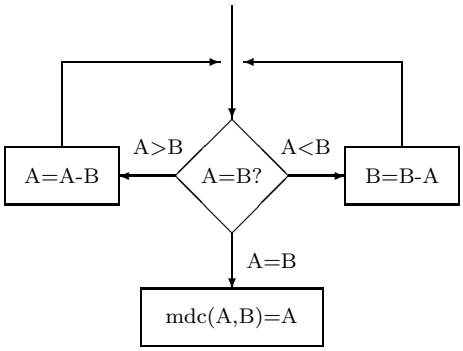
```
/* Exemplo que demonstra o uso da instrucao do...while */
#include <stdio.h>

void main ()
{
  int contagem = 1;
  /* Imprime na tela os numeros de 1 a 10 */
  do {
    printf("%d",contagem);
    contagem++;
  }while ( contagem <= 10 );
}
```

A saída será: **1 2 3 4 5 6 7 8 9 10**

5.6 Exercício

1. Faça um programa em C para ler números inteiros e determinar se o número lido é primo, imprimindo o resultado. A execução encerra quando o número lido for zero, e o programa deve então imprimir o maior e o segundo maior número primo lido.
2. Faça um programa com uma função para calcular a série de Fibonacci de um número **n** qualquer.

   A série de Fibonacci (F) é definida da seguinte maneira:
   - F(1) = 1
   - F(2) = 1
   - F(n) = F(n-1)+F(n-2)
3. Escreva um programa em C para calcular o máximo divisor comum e o mínimo múltiplo comum entre dois números. Abaixo está o diagrama do algoritmo que calcula o mdc:



4. Escreva um programa em C que seja capaz de converter um número de decimal para binário e de binário para decimal.

Capítulo 6

Controle de fluxo do programa

6.1 Instrução *break*

A instrução *break* pode ser colocada dentro de uma repetição *(for, while ou do...while)* ou ainda dentro de um *switch* (veja mais adiante).

Quando a instrução *break* é encontrada dentro de uma repetição, a execução da repetição na qual ele se encontra é encerrada.

```
for(contagem = 0; contagem < 10; contagem++)
{
  if ( contagem == 5 ) /* quando contagem for 5 */
    break; /* interrompe o loop */
}
```

6.2 Instrução *continue*

Como a instrução *break*, a instrução *continue* também só pode ser colocada dentro de um "loop"*for, while ou do...while*. Quando uma instrução *continue* é executada, a próxima iteração do loop começa imediatamente, ou seja, as instruções existentes entre a instrução *continue* e o final do "loop"não são executadas.

6.3 Instrução *return*

A instrução *return* termina a execução da função em que se encontra o programa e faz com que a execução continue na instrução seguinte à chamada da função. Esta instrução aceita um único argumento, que pode ser qualquer expressão válida em C, cujo valor é retornado.

Exemplo:

```
int MAX( int a, int b )
{
  if ( a > b )
    return(a);
  else
    return(b);
}
```

6.4 Função *exit()*

A função *exit()*, que pertence à biblioteca padrão *stdlib.h*, termina a execução do programa e retorna o controle ao sistema operacional. Esta função aceita um único argumento do tipo int, que é passado de volta ao sistema operacional para indicar o sucesso ou fracasso do programa.

6.5 Instrução *switch*

A instrução *switch* é semelhante à instrução if , ou seja, a partir da avaliação de uma expressão a instrução *switch* pode realizar diferentes ações e, ao invés do *if*, não está restrita a apenas duas ações. A forma geral é a seguinte:

```
switch (expressão) {
  case gabarito1 :  instruções;
  break;
  case gabarito2 :  instruções;
  break;
    ...
  case gabaritoN : instruções;
  break;
  default:  instruções;
};
```

Nesta instrução, *expressão* é qualquer expressão que resulte em um valor inteiro do tipo *long, int*, ou *char*. A instrução *switch* avalia a expressão e compara o seu valor com os gabaritos após cada item *case*; então:

- Se for encontrada uma equivalência entre expressão e um dos gabaritos, a execução é transferida para as instruções subseqüentes ao item *case*.
- Se nenhuma equivalência for encontrada, a execução é transferida para as instruções subseqüentes ao item *default*, que é opcional.
- Se nenhuma equivalência for encontrada e não houver um item *default*, a execução é transferida para a primeira instrução subseqüente à chave de encerramento da instrução *switch*.

Observe que ao término de cada item *case* aparece uma instrução *break*. Na verdade o *break* é opcional. Sua funcionalidade é a mesma quando utilizado dentro de repetições, ou seja, o *break* faz com que a instrução *switch* seja terminada.

Se ao final do corpo de comandos de um case não houver um *break*, todos os gabaritos abaixo serão executados até que seja encontrado um *break* ou seja atingido o final do *switch*.

```
/* Exemplo do uso da instrucao switch */
#include <stdio.h>
#include <stdlib.h>

void main()
{
  char vogal;
  printf("\nDigite uma vogal:  ");
  scanf ("%c",&vogal); /* le uma letra */
  switch (vogal) {
    case 'a':  vogal='e'; break;
    case 'e':  vogal='i'; break;
    case 'i':  vogal='o'; break;
    case 'o':  vogal='u'; break;
    case 'u':  vogal='a'; break;
    default :  printf("Erro!A letra nao e vogal");
    exit(0);
  };
  printf ("a vogal subseqüente e %c \n",vogal);
}
```

6.6 Exercício

Esse exercício consiste em desenvolver uma pequena calculadora, com várias opções que serão selecionadas em um menu.

O menu principal deverá ser o seguinte:

1. Soma n números
2. Multiplica n números
3. Divide a/b
4. Subtração de n números
5. Calcular a^b
6. Calcular a média aritmética de n números
7. Sair

Cada opção deverá comportar-se da seguinte forma:

- requisitar quantos números farão parte da operação (quando aplicável);
- requisitar cada entrada;
- ao final das iterações, imprimir o resultado e aguardar uma tecla;
- retornar ao menu principal.

Capítulo 7

Funções

Uma função é uma seção independente de código em C, designada por um nome, que realiza uma tarefa específica e, opcionalmente, retorna um valor ao programa que a chamou.

7.1 Protótipo

O protótipo da função fornece ao compilador uma descrição de uma função que será definida posteriormente no programa. O protótipo inclui o tipo de retorno, que indica o tipo de variável que a função retornará. Além disso, ele inclui também o nome da função, os tipos de variáveis dos argumentos que serão passados para a função e opcionalmente o nome dos argumentos. O protótipo sempre termina com ';'.

```
tipo_de_retorno nome_funcao (tipo arg1,..., tipo argN);
```

7.2 Definição

A definição da função é a função propriamente dita. A definição contém as instruções que serão executadas. A primeira linha de uma definição de função, chamada de cabeçalho da função, deve ser idêntica ao protótipo da função, com exceção do ponto-e-vírgula final. Os nomes das variáveis usadas como argumentos devem necessariamente ser incluídos no cabeçalho. A seguir vem o corpo da função, contendo as instruções que a função executará contidas num bloco. Se o tipo de retorno da função não for *void*, uma instrução *return* deve ser incluída para retornar um valor compatível com o tipo de retorno especificado.

```
tipo_de_retorno nome_funcao (tipo arg1,..., tipo argN)
{
  instruções;
}
```

7.2.1 Tipo de retorno

Especifica o tipo de dado que a função deverá retornar ao programa que a chamou. O tipo pode ser qualquer dos tipos válidos em C ou um tipo definido pelo programador. A função pode não retornar nenhum valor.

```
int func1(...)  /* retorna um tipo int */
void func2(...)  /* não retorna dados */
```

Para retornar um valor a partir de uma função, é usada a palavra-chave *return* seguida por uma expressão válida em C. Quando a execução atinge a instrução *return*, a expressão é avaliada e o valor é transmitido para o programa que originou a chamada. O valor de retorno de uma função é o valor da expressão.

```
int func1(...)
{
  int x; /* declaracao da variavel */
  ...  /* demais instrucoes */
  ...
  return x*2; /* retorna x*2 */
}
```

7.2.2 Nome da função

Pode ser qualquer nome, contanto que siga as regras adotadas para atribuir nomes de variáveis em C. O nome da função deve ser único e é sempre aconselhável usar um nome que reflita a finalidade da função.

7.2.3 Corpo da função

É delimitado por chaves e colocado imediatamente após o cabeçalho da função. Quando uma função é chamada, a execução começa no início do corpo e termina (retornando ao programa de origem) quando uma instrução *return* é encontrada ou quando a execução chega ao final do bloco.

Podemos declarar variáveis dentro do corpo de uma função. Estas variáveis são chamadas de variáveis locais, significando que são privativas desta função e distintas de outras variáveis com o mesmo nome que possam ter sido declaradas em outras partes do programa. A declaração destas variáveis deve ser feita antes de qualquer instrução.

7.3 Chamada de funções

7.3.1 Chamadas por valor

Quando uma variável é passada para uma função pelo valor, a função tem acesso ao valor da variável, mas não à própria variável original. Portanto, as instruções contidas na função não podem modificar o valor da variável original.

7.3.2 Chamadas por referência

Neste caso a função recebe o endereço de memória (uma referência) do parâmetro, ao invés do valor do parâmetro. Desta forma, a função pode alterar o valor da variável mediante a utilização deste endereço. Tanto a função quanto o programa devem ser informados de que um parâmetro é chamado por referência utilizando os operadores **&** (para passar um endereço) e ***** (para receber um endereço).

Exemplo:

```
/* Passando argumentos por valor e por referência */

void por_valor(int a, int b, int c);
void por_ref(int *a, int *b, int *c);

void main ()
{
  int x = 2, y = 4, z = 6;

  printf("\nAntes de chamar por_valor():x=%d, y=%d, z= %d",x,y,z);
  por_valor(x, y, z);

  printf("\nDepois de chamar por_valor():  x=%d, y=%d, z=%d",x,y,z);
  por_ref(&x,&y,&z);

  printf("\nDepois de chamar por_ref():  x=%d, y=%d, z=%d",x,y,z);
}
```

```
void por_valor(int a, int b, int c)
{
  a = b = c = 0;
}

void por_ref(int *a, int *b, int *c)
{
  a = *b = *c = 0;
}
```

A saída será: **Antes de chamar por_valor(): x = 2, y = 4, z = 6**
Depois de chamar por_valor(): x = 2, y = 4, z = 6
Depois de chamar por_ref(): x = 0, y = 0, z = 0

Obs.:
& - Passa o endereço de memória da variável passada por parâmetro e, portanto, as modificações sobre esta variável realizadas na função são permanentes.
* - Acessa o conteúdo de tal endereço de memória.

7.4 Exemplo de função

```
/* Calcula a divisao de dois numeros */
#include <stdio.h>
#include <stdlib.h>

float divisao(float a, float b); /* Prototipo da função */

void main()
{
  float x = 13.75, y = 1.38;
  printf("O valor da divisao de %f por %f e:  %f.",x,y,divisao(x,y));
}

float divisao(float a, float b) /* Definicao da funcao */
{
  float div;

  if (b != 0) /* Se b for nao nulo */
  {
    div = a / b; /* faz a divisao */
    return(div);
  }
  else /* Se b for nulo */
  {
    /* Imprime mensagem de erro e aborta execucao */
    printf("\nDivisao por zero!");
    exit(1);
  }
}
```

7.5 Exercícios

1. Transforme o programa que detecta se um número é primo em uma função.
2. Escreva uma função que gera e imprime os primeiros n números primos, sendo que n é o parâmetro desta função.

Capítulo 8

Matrizes

Uma matriz é uma coleção de localizações para armazenamento de dados, todas contendo o mesmo tipo de dados e acessadas pelo mesmo nome. Cada localização de armazenamento da matriz é chamada de *elemento da matriz*.

8.1 Matrizes Unidimensionais

Uma matriz unidimensional é uma matriz que possui um único subscrito. Um *subscrito* é um número entre colchetes colocado após o nome da matriz. Esse número é usado para identificar os elementos individuais de uma matriz.

Quando uma matriz é declarada, o compilador reserva um bloco de memória com tamanho suficiente para conter todos os seus elementos. Os elementos individuais da matriz são armazenados seqüencialmente na memória.

8.2 Matrizes Multidimensionais

Uma matriz multidimensional tem mais de um subscrito. Uma matriz bidimensional tem dois subscritos, uma matriz tridimensional tem três subscritos e assim por diante. Não há qualquer limite ao número de dimensões que uma matriz pode ter em C.

8.3 Inicialização de Matrizes

Uma matriz pode ser total ou parcialmente inicializada no momento em que é declarada. Para fazer isto, basta colocar um sinal de igualdade após a declaração da matriz e acrescentar uma lista de valores entre chaves, separados por vírgulas. Estes valores serão atribuídos pela ordem aos elementos da matriz.

Exemplos:

```
int matriz[4] = { 100, 200, 300, 400 }
/* equivale a:
matriz[0] = 100; matriz[1] = 200;
matriz[2] = 300, matriz[3] = 400;
*/

int matriz[ ] = { 100, 200, 300, 400 }
/* equivale a:
matriz[0] = 100; matriz[1] = 200;
matriz[2] = 300, matriz[3] = 400;
*/

int matriz[3][2] = { 1, 2, 3, 4, 5, 6 }
/* equivale a:
```

```
matriz[0][0] = 1;  matriz[0][1] = 2;
matriz[1][0] = 3;  matriz[1][1] = 4;
matriz[2][0] = 5;  matriz[2][1] = 6;
*/

/* O que seria equivalente a: */
int matriz[3][2] = { {1, 2}, {3, 4}, {5, 6} }
```

8.4 Referenciando um elemento na matriz

Quando referenciamos um elemento da matriz, o subscrito (número que colocamos entre colchetes seguindo o nome da matriz) irá especificar a posição do elemento na mesma.

Os elementos da matriz são sempre numerados por índices iniciados em 0 (zero). Então, o elemento referenciado pelo número 2 não será o segundo elemento da matriz, mas sim o terceiro. Assim, o último elemento de uma matriz de tamanho N é referenciado por N-1.

8.5 Lendo um número desconhecido de elementos

Quando não sabemos de antemão quantos itens entrarão em nossa matriz, podemos usar uma constante simbólica para definir o tamanho da matriz na declaração. Assim o tamanho da matriz poderá ser facilmente modificado.

Exemplo:

```
#define LIM 30 /* define um tamanho p/ a matriz */

void main()
{
  /* declara uma matriz de tamanho LIM */
  int matriz[LIM];
  /* declara um contador e inicia-o com zero */
  int i = 0;

  do
  {
    printf("Entre com um numero:");
    scanf("%d", &matriz[i]); /* le um numero */
    i++;
  }while (i < LIM);
}
```

8.6 Exercício

1. Crie uma função que receba como parâmetros um vetor e o seu tamanho, e que ordene esse vetor. A função não deverá retornar nenhum valor.

2. Crie uma função que recebe como parâmetros um vetor, o seu tamanho e um valor X e que retorne a primeira posição do vetor cujo valor é igual ao de X. Se a função não encontrar o elemento procurado, ela deve retornar -1.

3. Escreva um programa em C para gerar cartões de loteria. Ele recebe como entrada o número de cartões e a quantidade de números por cartão, além do limite inferior e superior destes números. Ao final, o programa deve imprimir estes cartões. (DICA: para gerar números aleatórios, utilize a função *random()* da biblioteca *stdlib.h*).

4. Escreva um programa em C que calcula a determinante de uma matriz 3x3.

5. Escreva um programa em C que calcula a multiplicação entre duas matrizes R e S, sendo R de dimensão i $\times$ j, e S de dimensão j $\times$ k.

Capítulo 9

STRINGS

Uma *string* é uma seqüência de caracteres. As *strings* são usadas para conter dados de texto - consistem de letras, algarismos, marcas de pontuação e outros símbolos.

Na linguagem C, *strings* são armazenadas em matrizes do tipo *char*, e obedecem às mesmas regras válidas para estas.

Exemplos:

```
char string[7] = {'B', 'r', 'a', 's', 'i', 'l', '\x0'};
char string[7] = "Brasil";
char string[ ] = "Brasil";
```

OBS.: O símbolo "\x0"indica o final de uma string.

9.1 Algumas funções de manipulação de string

A linguagem C oferece uma biblioteca padrão - *string.h* - para manipulação de strings. Apresentamos a seguir algumas das funções contidas nesta biblioteca.

9.1.1 strcpy()

Copia toda a segunda string para a primeira e retorna a string copiada.

```
char *strcpy( char *destino, char *origem );
```

9.1.2 strlen()

Retorna o tamanho de uma string.

```
unsigned long strlen( char *string );
```

9.1.3 strcat()

Concatena uma cópia da segunda string ao final da primeira e retorna a string concatenada.

```
char *strcat( char *str1, char *str2 );
```

9.1.4 strcmp()

Compara duas strings, caractere por caractere, e retorna um valor.

Uma observação importante é que os caracteres minúsculos são alfabeticamente maiores que os caracteres maiúsculos no contexto da linguagem C. Isso acontece devido ao uso da tabela **ASCII**.

```
int strcmp( char *str1, char *str2 );
```

Valor de Retorno	Significado
< 0	str1 é alfabeticamente menor do que str2
$= 0$	str1 é igual a str2
> 0	str1 é alfabeticamente maior do que str2

9.2 Exercícios

1. Escreva um programa em C que receba várias palavras (strings) e as imprima na tela em ordem alfabética. Invente uma maneira criativa para o usuário indicar ao programa que não há mais palavras a serem lidas.

2. Escreva um programa em C que lê uma *string* do teclado e imprime a string invertida.

3. Um palíndromo é uma palavra tal que quando revertida é igual a ela mesma. Escreva um programa que recebe uma palavra e responde se ela é palíndromo ou não.

4. Para toda palavra é possível obter um subconjunto de letras consecutivas da mesma. Este conjunto pode receber denominações específicas de acordo com algumas propriedades que este respeita. Veja a seguir:

   Se P é a palavra completa e S é um subconjunto de caracteres consecutivos de P e o operador + representa a concatenação de duas strings, então:

   - **prefixo**
     S será *prefixo* de **P** se $P = S + S'$, onde **S'** é um outro conjunto de caracteres.
   - **sufixo**
     S será *sufixo* de **P** se $P = S' + S$, onde **S'** é um outro conjunto de caracteres.
   - **substring**
     S será *substring* se não estiver de acordo com nenhuma das condições anteriores.

   De posse desses conceitos, escreva um programa em C que requisita uma *string* P qualquer, a armazena, e logo depois requisita a entrada de uma *string* S e imprime na tela se S é prefixo, sufixo ou substring de P.

   O resultado será tal como o exemplo abaixo:

```
avaliação
val
‘‘val com relação a avaliação é:’’
prefixo:  não
sufixo:  não
substring:  sim

cachorro

‘‘ com relação a cachorro é:’’
prefixo:  sim
sufixo:  sim
substring:  sim
```

Capítulo 10

Estruturas

Uma estrutura é uma coleção de uma ou mais variáveis agrupadas sob um único nome para facilitar a sua manipulação. As variáveis de uma estrutura, ao contrário das variáveis de uma matriz, podem ser de diferentes tipos de dados. Uma estrutura pode conter quaisquer tipos de dados válidos em C, inclusive matrizes e até mesmo outras estruturas.

10.1 A palavra-chave struct

```
struct rotulo{
  tipo campo1;
  ...
  tipo campoN;
} [instância];
```

A palavra-chave *struct* é usada para declarar estruturas em C. Ela identifica o início de uma definição de estrutura e é sempre seguida por um rótulo, que é o nome atribuído à estrutura. A seguir vêm os membros da estrutura, delimitados entre chaves. Uma instância, que é uma declaração de uma variável do tipo da estrutura, também pode ser incluída na definição.

10.2 Definindo e declarando

Existem basicamente duas maneiras de definir e declarar estruturas em C:

- A primeira é colocar uma lista com um ou mais nomes de variáveis imediatamente após a definição da estrutura, conforme o exemplo a seguir:

  ```
  struct coord { /* definicao de uma estrutura */
    int x;
    int y;
  } primeira, segunda; /* declaracao de variaveis */
  ```

  Estas instruções definem o tipo de estrutura coord e declaram duas estruturas (variáveis) deste tipo, chamadas primeira e segunda.

- A segunda maneira é declarar variáveis da estrutura em algum outro ponto do código-fonte. Por exemplo:

  ```
  struct coord { /* definicao de uma estrutura */
    int x;
    int y;
  };

  /* instrucoes adicionais aqui...  */
  ```

```
/* declaração de variaveis */
struct coord primeira, segunda;
```

10.2.1 Acessando os membros de uma estrutura

Os membros individuais de uma estrutura podem ser usados como qualquer outra variável respeitando-se as características do seu tipo. Para acessar os membros de uma estrutura, utiliza-se o operador de membro de estrutura '**.**' entre o nome da estrutura e o nome do membro.

Exemplo:

```
struct coord {
  int x;
  int y;
} primeira, segunda;

primeira.x = 50;
primeira.y = -30;
```

Uma das vantagens de se utilizar estruturas ao invés de variáveis individuais é a capacidade de copiar informações entre estruturas do mesmo tipo através de uma única instrução de atribuição. Continuando com o exemplo anterior, a instrução:

```
segunda = primeira;
```

é equivalente a:

```
segunda.x = primeira.x;
segunda.y = primeira.y;
```

Exemplo:

Recebe informações sobre as coordenadas dos cantos de um retângulo e calcula a sua área. Presume que a coordenada y no canto superior esquerdo é maior que a coordenada y no canto inferior direito, que a coordenada x do canto inferior direito é maior do que a coordenada x do canto superior esquerdo, e que todas as coordenadas são positivas.

```
#include <stdio.h>

struct coord{
  int x;
  int y;
};

struct retang{
  struct coord esqcima;
  struct coord dirbaixo;
};

void main ()
{
  int altura, largura;
  long area;
  struct retang quadro;

  /* recebe as coordenadas */
  printf("\nDigite a coordenada x superior esq.:");
  scanf("%d", &quadro.esqcima.x);

  printf("\nDigite a coordenada y superior esq.:");
```

```
  scanf("%d", &quadro.esqcima.y);

  printf("\nDigite a coordenada x inferior dir.:");
  scanf("%d", &quadro.dirbaixo.x);

  printf("\nDigite a coordenada y inferior dir.:");
  scanf("%d", &quadro.dirbaixo.y);

  /* calcula o comprimento e a largura */
  largura = quadro.dirbaixo.x - quadro.esqcima.x;
  altura = quadro.esqcima.y - quadro.dirbaixo.y;

  /* calcula e informa a area */
  area = largura * altura;
  printf("\nO retangulo tem uma area de %ld unidades.", area);
}
```

A saída será:

Digite a coordenada x superior esq.: 1
Digite a coordenada y superior esq.: 1
Digite a coordenada x inferior dir.: 10
Digite a coordenada y inferior dir.: 10
O retângulo tem uma área de 81 unidades.

OBS.: Quando estruturas são passadas por referência, utilizamos o operador '->' ao invés do operador '.'.

Exemplo:

```
/* Passando argumentos por valor e por referencia */

struct coord{
  int x;
  int y;
};

void por_valor( struct coord );
void por_ref( struct coord * );

void main ()
{
  struct coord ponto;

  ponto.x = 2;
  ponto.y = 4;

  printf("\nAntes de chamar por_valor():  x = %d, y = %d",
      ponto.x,ponto.y);

  por_valor(ponto); /* chamada por valor */

  printf("\nDepois de chamar por_valor():  x = %d, y = %d",
      ponto.x,ponto.y);

  por_ref(&ponto); /* chamada por referencia */

  printf("\nDepois de chamar por_ref():  x = %d, y = %d",
      ponto.x,ponto.y);
```

```
}

void por_valor(struct coord ponto)
{
  ponto.x = ponto.y = 0;
}

void por_ref(struct coord *ponto)

{
  ponto->x = ponto->y = 0;
}
```

A saída será:

Antes de chamar por_valor(): x = 2, y = 4
Depois de chamar por_valor(): x = 2, y = 4
Depois de chamar por_ref(): x = 0, y = 0

10.3 Exercício

Crie um pequeno banco de dados que armazenará os seguintes dados de um aluno:

- código de matrícula
- nome
- telefone
- endereço

O programa deverá ter um menu com as seguintes opções:

- inserir aluno
- imprimir alunos
- ordenar alunos

Cada aluno inserido será armazenado em um vetor de tamanho TAM_MAX e o programa deverá acusar erro se houver tentativa de inserção de alunos, caso o vetor esteja cheio.

No caso da função de ordenação, esta deverá ser realizada pela ordem alfabética do nome.

Capítulo 11

Especificadores e modificadores de tipos

11.1 Casting

Na linguagem C existem basicamente duas maneiras de converter uma variável de um tipo para outro: *implicitamente* ou *explicitamente*. Uma conversão implícita é aquela que ocorre quando é realizada a atribuição de uma variável de um determinado tipo para outra variável de um tipo diferente. O exemplo abaixo ilustra uma conversão implícita.

```
int x;
float y=10.53;
x=y;  ⇐  aqui ocorre uma conversão implícita
```

Nem sempre a linguagem C aceita conversões implícitas, principalmente se os tipos de dados envolvidos forem muito diferentes.

```
struct prod {
  int code;
  char nome[15];
};
struct prod var_b;
int i;
i = var_b;  ⇐  atribuição inválida!
```

Uma conversão explícita é quando o programador especifica qual o tipo que determinada variável deve assumir, por exemplo, através de um *casting*.

```
int x;
float y=10.53;
x=(int)y;  ⇐  o parênteses com o tipo é um casting
```

Assim como a conversão implícita, conversões explícitas entre tipos de dados muito diferentes também não são válidas.

Basicamente, a única vantagem de um casting sobre uma conversão implícita é para determinar exatamente qual o tipo de dado que uma determinada variável deve assumir.

Casting mudando resultados

```
int a, b;
float c,d;

a=5; b=2;
c=a/b;  ⇐  c divisão entre inteiros
d=(float)a/(float)b;  ⇐  d divisão entre floats
```

No exemplo anterior, embora a variável *c* seja do tipo *float*, as variáveis *a* e *b* são inteiras, portanto o resultado da divisão será **2** e não *2,5*.

Isso acontece porque operações que envolvem dois números inteiros sempre resultam em um valor inteiro.

Por outro lado, a variável *d* assumirá o valor *2.5* porque foi especificado, através de *casting*, que as variáveis *a* e *b* deveriam ser consideradas como *float*, o que obriga o programa a calcular a divisão entre dois float's ao invés de dois números inteiros.

11.2 Variáveis Static

Variáveis *static* são variáveis cujo valor é preservado entre sucessivas chamadas à função em que foi declarada. Elas diferem das variáveis locais porque o seu valor não é destruído quando a função termina e diferem das variáveis globais porque somente são visíveis dentro do contexto da função em que foram declaradas. Observe o código abaixo:

```
#include <stdio.h>
int prox_par(void)
{
  static int pares = -2;
  pares += 2;
  return (pares);
}

void main(void){
  int i;
  printf ("Os numeros pares menores que 100 sao:  ");
  for(i=0; i<100; i++)
    printf ("%d", prox_par());
}
```

Este programa imprime na tela números pares, iniciando por zero, enquanto a variável *i* for menor que 100, utilizando apenas a sucessiva chamada da função *prox_par*.

O segredo está na variável *static* declarada. Ela é inicializada com o valor -2 quando o programa inicia a execução. Na primeira chamada à função, essa variável é acrescida de 2, resultando no valor 0, que é devolvido pela função.

Na próxima chamada à função, o valor antigo da variável (zero) é preservado. Ela é então acrescida de 2, o valor 2 (0+2=2) é retornado e assim sucessivamente. Observe que se a variável não fosse declarada como *static*, a função retornaria zero sempre.

11.2.1 Exercício

Escreva uma função capaz de armazenar uma string e imprimir seu conteúdo. A função deverá ter apenas um parâmetro *(int p)* e funcionará de modo que se p=0 a função obtêm e armazena a string e se p=1 a função deverá apenas imprimir a string armazenada.

11.3 Variáveis Register

Este modificador de tipo é tradicionalmente aplicado a variáveis do tipo *int* e *char*, embora possa ser aplicado a outros tipos de dados também.

O modificador *register* requisita ao compilador que mantenha, se possível, o valor das variáveis declaradas como *register* em registradores da CPU, ao invés de mantê-las na memória.

Isso significa que as operações com tais variáveis são muito mais rápidas, pois o valor já está na própria CPU, economizando assim acessos à memória. É importante observar que o ganho de desempenho só será significativo quando aplicado a variáveis muito utilizadas, como índices de matrizes e controladores de repetições.

Atualmente este modificador não é muito utilizado e tende a se tornar obsoleto, pois os compiladores modernos vêm com recursos que analisam e otimizam automaticamente o código, de forma muito mais eficiente (e rápida) que qualquer ser humano.

Exemplo de declaração:

```
register int contador;
```

11.4 typedef

A linguagem C permite definir explicitamente novos nomes de tipos de dados usando-se a palavra reservada *typedef*. Não é criado um novo tipo de dado, apenas definido um novo nome (apelido) para um tipo de dado existente.

A forma geral da declaração *typedef* é:

```
typedef <tipo> <identificador>;
```

onde *tipo* é qualquer tipo de dado válido e *identificador* é o novo nome para esse tipo. O novo nome definido não substitui o nome do tipo, apenas é usado como um apelido para o mesmo.

Por exemplo:

```
typedef float flutuante;
...
flutuante atraso;
```

O *typedef* é extremamente útil para simplificar nomes muito longos e complexos, como por exemplo, nomes de estruturas.

Por exemplo:

```
typedef struct cliente_tipo {
  float divida;
  int atraso;
  char nome[40];
} cliente;
cliente lista[50];
```

Nesse exemplo, *cliente* não é uma variável do tipo *cliente_tipo*, mas um outro nome para *struct cliente_tipo*.

O código a seguir é equivalente ao anterior.

```
struct cliente_tipo {
  float divida;
  int atraso;
  char nome[40];
};
typedef struct cliente_tipo cliente;
cliente lista[50];
```

11.5 Campos de bit (bit fields)

A linguagem C permite que sejam acessados os bits individuais dentro de um tipo de dados maior como, por exemplo, um byte.

Campos de bit são campos de uma estrutura formatados de modo a ocuparem uma quantidade de bits definível pelo programador.

Essa propriedade é muito útil quando se deseja definir estruturas que vão conter dados que, normalmente, precisariam de menos que 8 bits (menor tamanho possível para um tipo de dado comum), ou mesmo um dado que ocuparia uma quantidade de bits que não seja múltiplo de 8.

A forma genérica para se definir um campo de bit é:

```
struct nome_estrutura{
  tipo_var var :  nbits;
  ...
};
```

Onde, além dos elementos tradicionais de uma estrutura, existe um novo elemento denominado *nbits*, que determina o tamanho, em bits, que a variável deve ocupar.

Observe que nbits não pode ser maior que o tamanho normal do tipo de dado *tipo_var*.

```
struct pessoa {
  char nome [30];
  int sexo:  1; /* 0-> feminino , 1-> masculino */
  int idade:  7;
  int estado_civil:  2;
  /* 0-> solteiro
    1-> casado
    2-> desquitado
    3-> viuvo */
} pessoas[2];

int main ()
{
  pessoas[0].nome = "Jose Maria da Silva";
  pessoas[0].sexo = 1; /* masculino */
  pessoas[0].idade = 34;
  pessoas[0].estado_civil = 0; /* solteiro */
}
```

11.6 Exercícios

1. Defina uma estrutura para armazenar os seguintes campos:

   - produto: [10 caracteres]
   - pago: (valores que pode assumir: s/n)
   - código: (valores que pode assumir: 0-31)
   - setor: (valores que pode assumir: 0-3)

   Defina duas estruturas, uma sem campos de bits e outra com campos de bits de modo a ocupar o menor tamanho possível.

   Baseado na tabela abaixo, compare a diferença entre os tamanhos de uma estrutura normal (sem campos de bits) e uma estrutura com campos de bits.

Tipo de Dado	**tamanho(em bytes)**
char	1
int	4

Tabela 11.1: Tamanhos de Dados

Capítulo 12

Operadores de bit

12.1 O que é uma operação bit-a-bit

Uma operação bit-a-bit é, basicamente, uma operação lógica aplicada sobre cada bit de um número inteiro.

Entende-se por operações lógicas *AND, OR, NOT, XOR e SHIFT's.*

Exemplo:

Tome os números 5 e 14. Em binário:

$5 \Rightarrow 0101, 14 \Rightarrow 1110$

Uma operação AND bit-a-bit entre 5 e 14:

$$\begin{array}{r} 0101 \\ 1110 \\ \hline 0100 \end{array}$$

Ou seja, **5 AND 14 = 4**

Observe que a operação *AND* bit-a-bit realiza um *AND* entre os pares de bit respectivos de cada operando.

A linguagem C suporta um conjunto completo de operadores de bit que podem ser aplicados apenas aos tipos de números inteiros (*char, int e long*).

Os operadores de bit permitem que programadores possam escrever instruções de alto nível que operam em detalhes de baixo nível. Por exemplo, controlar registradores de hardware ligando e desligando bits específicos em certos endereços de memória onde estes registradores estão localizados logicamente.

12.2 Representação de números hexadecimais

Para os computadores, todos os dados e informações são representados em binário(base 2), porque esta representação simplifica o projeto do hardware.

O grande problema com esta representação é que para seres humanos os dados ficam difíceis de serem compreendidos e representados, já que mesmo números relativamente pequenos precisam de uma quantidade muito grande de bits para serem representados.

A princípio poderia-se pensar em transformar estes números em binário em representação decimal (base 10), já que é a representação mais natural para o ser humano. O problema de usar a base decimal para representar números binários é que o processo de conversão é relativamente lento e altamente sujeito a erros.

Para resolver estes problemas, a base adotada para representar dados (pelo menos em baixo nível) é a base hexadecimal (base 16) porque representa números binários de maneira mais inteligível e a conversão entre a base 2 e a base 16 é bastante fácil e direta.

Na linguagem C, para especificar que uma determinada constante numérica está na base hexadecimal, basta iniciar a representação do número com 0x (zero x).

Exemplo:

0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1

Ex:

	Dec.	Hex.	Binário
	31503	0x7b0f	0111 1011 0000 1111
&	16277	0x3f95	0011 1111 1001 0101
	15109	0x3b05	0011 1011 0000 0101

Figura 12.1:

`int z=0xff;` ⇐ atribui a constante $(\text{ff})_{16}$ ao z.

12.3 Operações bit-a-bit em C

Operador	Descrição
`&`	E binário (AND)
`|`	OU binário (OR)
`^`	OU exclusivo binário (XOR)
`<<`	Desloca bits para a esquerda (LEFT SHIFT)
`>>`	Desloca bits para a direita (RIGHT SHIFT)
`~`	Complemento (nega todos os bits)

Tabela 12.1: Operadores de bits existentes em C

Em expressões, os operadores *AND* (&), *OR* (|) e *XOR* (^) combinam valores de acordo com as regras para seus equivalentes lógicos. Deve-se tomar cuidado para não confundir os operadores de bit & e | com os operadores lógicos && e ||. Da mesma forma com o que acontece com os operadores = e ==, a linguagem C permite que usemos o operador errado em muitos casos sem apontar o erro.

12.4 Operador & (AND)

O operador & executa uma operação E (AND) lógica em cada um dos bits dos operandos. Observe o quadro na figura 12.1.

À esquerda temos a tabela-verdade para o operador & : o bit de resultado é 1 somente quando ambos os bits dos operandos forem 1. À direita é exibido um exemplo de uma operação *AND*. Tanto os operandos quanto o resultado são exibidos em três bases: decimal, hexadecimal e binária.

Vale a pena ressaltar que, quando bits são manipulados, a base decimal não é muito apropriada para a representação. Ao invés da base decimal, utiliza-se a base hexadecimal.

Um uso típico do operador & é mascarar parte de um número, ou seja, permitir que apenas uma parte desse número seja utilizada para um determinado fim.

Suponha que uma determinada variável *VAR_A* represente a configuração de um programa.

Cada bit desse número representa o estado ativado(1)/desativado(0) para uma determinada opção do programa.

Se deve ser escrito um algoritmo para detectar se a opção associada ao 3^o bit está ativada, pode ser utilizado um AND de bits da seguinte forma:

VAR_A & 4 ⇒ 4 em binário é $(100)_2$

O resultado desta operação só será diferente de zero se o 3^o bit estiver ativado.

12.5 Operador | (OR)

O operador | executa uma operação OU (OR) entre cada bit dos operandos. As regras lógicas que definem o operador | determinam que se pelo menos um dos bits for 1, então o bit do resultado será 1. No caso de ambos os bits serem 0, o resultado será 0.

O operador | pode ser utilizado para ligar (setar) bits em números.

0 | 0 = 0	Ex:		Dec.	Hex.	Binário
0 | 1 = 1			31503	0x7b0f	0111 1011 0000 1111
1 | 0 = 1		|	16277	0x3f95	0011 1111 1001 0101
1 | 1 = 1			32671	0x7f9f	0111 1111 1001 1111

Tabela 12.2: Regras para o operador |

Considerando novamente o exemplo da variável *VAR_A* que determina a configuração de um programa, suponha que seja necessário ativar o 3^o bit. Para tanto, a seguinte operação pode ativar o 3^o bit:

VAR_A | 4 $\Rightarrow$ 4 em binário é 100

Ou seja, como pelo menos o 3^o bit da constante 4 é 1, isso significa que o resultado da operação terá o 3^o bit com o valor 1.

12.6 Operador ^ (XOR)

O operador ^ executa uma operação OU EXCLUSIVO (XOR) entre cada um dos bits dos operandos. Genericamente, dados n operandos, o resultado de uma operação *XOR* entre os n operandos será 1 se, e somente se, o número de operandos que assumem valor 1 é impar.

0 ^ 0 = 0	Ex:		Dec.	Hex.	Binário
0 ^ 1 = 1			31503	0x7b0f	0111 1011 0000 1111
1 ^ 0 = 1		^	16277	0x3f95	0011 1111 1001 0101
1 ^ 1 = 0			17562	0x449a	0100 0100 1001 1010

Tabela 12.3: Regras para o operador ^

Na esquerda do quadro é exibida tabela-verdade que define a operação e à direita um exemplo mostrado nas três bases mais usadas.

A operação *XOR* possui uma propriedade interessante: se uma máscara é aplicada duas vezes ao mesmo número, o resultado é o número original. De uma forma genérica, podemos dizer que (A ^ B) ^ B = A.

A tabela 12.4 demonstra esta propriedade.

	Dec.	Hex.	Binário
	31503	0x7b0f	0111 1011 0000 1111
^	16277	0x3f95	0011 1111 1001 0101
	17562	0x449a	0100 0100 1001 1010
	17562	0x449a	0100 0100 1001 1010
^	16277	0x3f95	0011 1111 1001 0101
	31503	0x7b0f	0111 1011 0000 1111

Tabela 12.4: Propriedade especial do XOR

Esta propriedade torna a operação *XOR* bastante usada em criptografia. Aplica-se uma máscara sobre um documento através de um XOR e obtém-se o documento "disfarçado". Reaplicando-se a máscara, obtém-se de volta o documento original. Naturalmente, só isso não garante uma criptografia segura, mas muitos algoritmos sofisticados de criptografia ainda fazem uso dessa capacidade especial do *XOR* misturada com outras técnicas.

12.7 Operadores << e >>

Os operadores de deslocamento para a esquerda (shift left - <<) e de deslocamento para a direita (shift right - >>) movimentam (empurram) os bits para a esquerda e direita, respectivamente.

A forma geral de uso desses operadores é:
[valor] >> [número de posições]
[valor] << [número de posições]
Observe o exemplo abaixo:

```
/* ...  */
unsigned int x = 0xC743; /* x = 1100 0111 0100 0011 */
x = x << 3; /* x = 0011 1010 0001 1000 */
/* ...  */
```

Observe que os bits de x foram empurrados 3 posições para a esquerda. Os 3 bits mais significativos de x são perdidos e os 3 bits menos significativos passam a valer 0. No deslocamento para a direita, o processo é semelhante, porém os bits menos significativos é que são perdidos, enquanto zeros são introduzidos nos bits mais significativos.

Convém ressaltar que os deslocamentos não são rotações, ou seja, os bits que saem de um lado não retornam pelo outro.

Os operadores de deslocamento também podem ser usados para realizar multiplicação ou divisão rápida de inteiros. Um deslocamento de 1 bit à esquerda equivale a multiplicar o número por 2. Já um deslocamento de 1 bit para a direita equivale a dividir o número por 2. Genericamente, temos:

```
x = x << n; /* equivalente a:  x = x * 2n; */
x = x >> n; /* equivalente a:  x = x / 2n; */
```

12.8 Operador ~ (complemento)

O operador ~ (complemento) é um operador unário, isto é, trabalha com somente um operando, que é colocado após o símbolo do operador, assim como o operador ! (NOT).

A sua função é calcular o *complemento* do valor fornecido, ou seja, trocar todos os bits 0 por 1 e todos os bits 1 por 0. Assim como todos os outros operadores, o operador ~ não altera o valor do operando, o que significa que o resultado da operação deve ser atribuído a uma variável ou utilizado em uma expressão.

```
unsigned int v1, v2;
v1 = 0x7f4a; /* v1 = 0111 1111 0100 1010 */
v2 = ~v1; /* v2 = 1000 0000 1011 0101 */
```

12.9 Exercícios

1. Escreva um programa que imprime duas perguntas simples com resposta de múltipla escolha através de somatório. O programa deverá receber a resposta do usuário (que será um número) e deverá apresentar na tela a pontuação de acordo com o nível do acerto.

   O programa deverá imprimir o seguinte enunciado:

   Para responder as perguntas abaixo, verifique quais as alternativas que voce considera corretas, some o numero da alternativa e escreva essa soma como resposta.

   As perguntas e suas respectivas alternativas devem ser:

Quais funções abaixo pertencem a biblioteca stdio.h da linguagem C?
1) write
2) printf
4) getchar
8) cos
16) scanf
Resposta:[2+4+16=22]

Quais os tipos de dados são pré-definidos na linguagem C?
1) int
2) longint
4) long int
8) char
16) string
Resposta:[1+4+8=13]

Os números entre [ ] apresentam o somatório das opções corretas.

Os resultados deverão ser avaliados de acordo com a seguinte tabela:

opções corretas/total	**pontuação**	**texto a imprimir na tela**
1/3	1	Voce esta fraco.
2/3	3	Continue assim.
3/3	5	Parabéns!

Caso exista pelo menos uma opção incorreta, a pontuação deverá ser 0 e o programa deverá imprimir uma mensagem avisando que o usuário errou uma das respostas.

O valor digitado pelo usuário deverá ser comparado com um gabarito, que serão valores guardados em um pequeno vetor (de 2 posições).

2. Faça um programa que recebe um número(em decimal) e imprime o seu valor em binário. O programa só poderá utilizar operações lógicas.

3. Crie duas funções denominadas *rot_left (int s)* e *rot_right (int s)* que realizem as rotações, para esquerda e direita respectivamente, de seu parâmetro e retorne esse valor.

   Suponha que a rotação será sobre um número de 16 bits.

   Abaixo, ilustramos o efeito de rotações:

   var_x = 18 /* 18 é $(10011)_2$ em binário */
   rot_left(x) $\Rightarrow$ 00111
   rot_right(x) $\Rightarrow$ 11001

   Uma rotação empurra o bit que sumiria para o lado oposto.

Capítulo 13

Argumentos ARGV e ARGC

Algumas vezes é necessário que um programa receba argumentos através da linha de comando. Um argumento de linha de comando é o texto digitado após o nome de um programa executável (ex.: pkzip arquivo1 $\Rightarrow$ arquivo1 é um argumento do programa executável pkzip).

Para obter os argumentos da linha de comando, existem dois parâmetros que podem ser acrescentados à função main de um programa em C, *argc* e *argv*. O parâmetro *argc* armazenará o número de argumentos na linha de comando e é um inteiro. Ele será, no mínimo um, já que o nome do programa é qualificado como o primeiro argumento.

O parâmetro *argv* é uma matriz de strings. Todos os argumentos da linha de comando são considerados strings.

A forma de declaração será sempre a seguinte:

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

Observe que os nomes *argc* e *argv* são absolutamente arbitrários. Você poderá atribuir qualquer nome que preferir.

```
#include <stdio.h>
void main(int count, char *parametros[ ]){
  if (count != 2) {
    printf(``Voce esqueceu de digitar o seu nome!\n'');
    exit(0);
  }
  printf (``Alô %s'', parametros[1]);
}
```

Saída: Suponha que o programa acima se chama *alo* e foi chamado da seguinte forma:

```
alo Fulano
```

A saída será:

Alô Fulano.

Dada a seguinte linha de comando:

```
prog_exec arg1 arg2 arg3 ...  argn
```

Os argumentos *arg[1..n]* estarão, respetivamente, nas posições 1, 2, 3, ..n do vetor *argv*.

Portanto, *argv[1]* é o primeiro argumento passado na chamada do programa.

13.1 Retornando Valores da Função main()

Ainda que nenhum dos programas que você viu até agora tenha demonstrado, é possível retornar um valor inteiro a partir da função *main()*. Esse valor é retornado ao processo chamador, que usualmente é o sistema operacional. Você pode retornar um valor da função *main()* usando a declaração *return* como se faz com qualquer outra função.

A interpretação do valor de retorno de uma função depende do processo sobre o qual o programa é executado.

Exemplo:

```
/* COMLINE: um programa que executa qualquer comando que e
especificado na linha de comando.  Retorna um codigo de erro
para o sistema operacional se sua operacao falhar.  */

#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[ ]){
  int i;
  for (i=1; i<argc; i++)
  {
    if(system(argv[i]))
    {
      printf(``%s falha\n'', argv[i]);
      return -1; /* codigo de falha */
    }
  }
  return 0; /* retorna o codigo de sucesso da operacao */
}
```

Alguns programadores gostam de declarar especificamente a função *main()* como *void*, se ela não retorna um valor, usando uma declaração como esta:

```
void main(void);
```

Entretanto, isso não é necessário. Uma outra abordagem é sempre retornar um valor da função *main()*.

13.2 Exercício

Faça um programa que receba três argumentos na linha de comando. O primeiro argumento será algum operador aritmético(+,-,/,*) e o segundo e o terceiro argumento serão números.

O programa deverá imprimir na tela o resultado da operação do segundo argumento pelo terceiro.

Exemplo: Suponha que o programa seja chamado opera:

`opera + 2 3` $\Leftarrow$ deverá imprimir na tela 5

`opera * 5 5` $\Leftarrow$ deverá imprimir na tela 25

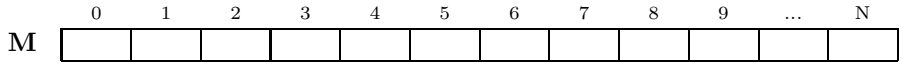
Capítulo 14

Ponteiros

Ponteiros são ferramentas extremamente versáteis e flexíveis disponibilizadas pela linguagem C para manipulação da memória. Através de ponteiros o programador tem controle praticamente total sobre o armazenamento de dados na memória.

14.1 O que são ponteiros?

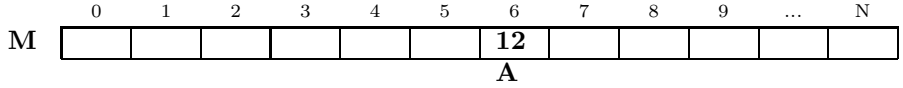
Para entender o que é uma ponteiro é necessário visualizar a memória como se fosse um grande e contínuo vetor. Vamos supor que a memória do computador é um vetor *M[n]*, como representado abaixo, **N+1** é o tamanho da memória:



Em um programa C, é possível declarar uma variável qualquer e uma região desta memória será automaticamente dedicada a esta variável. Por exemplo, considere a seguinte declaração:

```
int A;
```

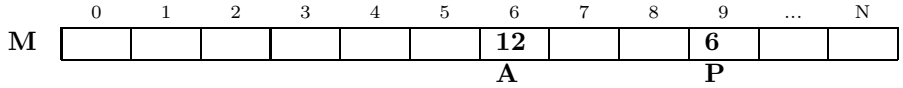
Quando o programa é executado, a declaração acima faz o sistema operacional *alocar*[1] uma região disponível da memória, que será apelidada de **A**. No exemplo, é ilustrada uma região da mémoria alocada.



Ou seja, no exemplo acima, o símbolo A é, na verdade, um apelido para a célula M[6].

Um ponteiro é uma variável que será utilizada como índice do vetor memória, ou seja, um ponteiro é uma variável que aponta para uma determinada posição de memória.

Por exemplo, suponha que tenha sido declarado um ponteiro de nome P.



A princípio, na ilustração acima, o ponteiro **P** tem a mesma aparência que a variável **A**, ou seja, é alocada também uma região de memória para que seu valor possa ser armazenado (*V[9]*) e a essa região denomina-se **P**.

A diferença entre um ponteiro e uma variável numérica tradicional é que um ponteiro é um tipo de dado pode ser utilizado para *indexar* a memória, o que não é permitido fazer com as outras variáveis.

Ou seja, o ponteiro **P** pode ser usado em uma forma que equivale a *M[**P**]*, ou seja, o seu conteúdo (que no exemplo é o valor 6) pode ser utilizado para obter o conteúdo de uma determinada posição de memória.

[1] Alocar uma memória para uma variável é obter uma região disponível da memória, de tamanho suficiente para conter os dados desta variável.

É importante ressaltar que a notação utilizada da memória como um vetor *M[ ]* é apenas para fins de ilustração. Não existe nenhum nome especial para denominar um vetor que é a memória.

14.2 Declarando ponteiros

Se um ponteiro aponta para uma determinada região de memória, faz sentido determinar qual é o tipo de dado armazenado na posição para a qual ele aponta.

Dessa forma, sempre que um ponteiro é declarado, deve ser especificado o tipo de dado para o qual ele aponta. Como um ponteiro pode apontar para qualquer tipo de dado, faz sentido uma sintaxe que visa simplificar a declaração.

Na linguagem C, para declarar um ponteiro basta acrescentar o símbolo '*' após o que seria uma declaração de tipo tradicional na linguagem C.

```
<tipo> *<nome-do-ponteiro>;
```

Exemplos de declaração de ponteiros:

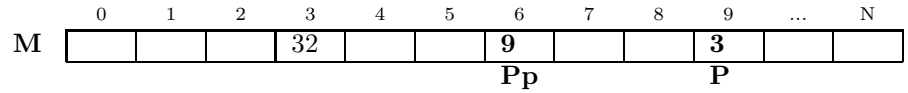
```
char *ptr1; /* ponteiro para um char */
int *ptr2; /* ponteiro para um int */
double *ptr3; /* ponteiro para um double */
struct registro{
  int campo1, campo2;
} *reg; /* ponteiro para uma estrutura do tipo registro */
```

Observe que um ponteiro também é considerado um tipo de dado, portanto não existe nenhuma restrição que impeça de se declarar um ponteiro que aponta para outro ponteiro.

Por exemplo:

```
int *P;
int **Pp;
```

Acima, foi declarado uma variável chamada *Pp* que aponta para uma região de memória cujo conteúdo é do tipo *int**. Essa situação é ilustrada abaixo:



Ou seja, *Pp* pode ser usado para indexar a memória *M* e obter o conteúdo da posição 9, que por sua vez também é um ponteiro e pode ser usado para indexar a memória *M* e obter o conteúdo da posição 3.

Equivalentemente, se *M* é um vetor, para obter o conteúdo da posição 3 a seguinte declaração seria suficiente:

M[M[**M**[**Pp**]]] $\Rightarrow$ M[**M**[**9**]] $\Rightarrow$ M[3]

14.3 Utilizando ponteiros

Até agora foi discutido sobre ponteiros, indexar a memória, mas também foi observado que não existe um vetor de memória *M* para que o ponteiro possa ser utilizado para indexar esse vetor.

Para indexar a memória utilizando o valor de uma variável que é ponteiro, basta preceder o nome da variável com o símbolo '*'. Neste contexto, o operador '*' é denominado *derreferenciador*. É necessário cuidado para não confundir o operador de **derreferenciação** (que é unário) com o operador de **multiplicação** (que é binário).

Por exemplo, dada a seguinte declaração:

```
int *P, A;
```

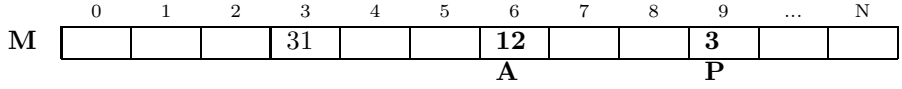
Para poder indexar a memória do valor de *P* e atribuir esse valor a variável *A*, é necessária a seguinte linha de código:

```
A = *P;
```
$\Rightarrow$ equivalente a: **A = M[P];**

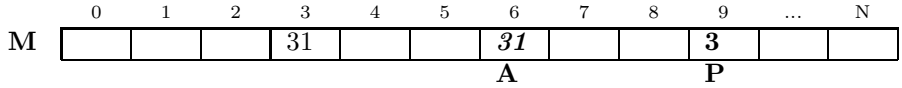
É importante diferenciar o símbolo **P* segundo o contexto em que ele se encontra. Toda vez que aparece um tipo de dado, um '*' e logo após um símbolo que nomeia a variável (i.e., `int *P`), isso não é

um acesso à posição de memória, é apenas uma declaração que denomina que P será um ponteiro para uma posição de memória cujo conteúdo é do tipo *int*.

Suponha que a memória está no seguinte estado:



Ao executar `A = *P`, a memória assumiria a seguinte configuração:



Ou seja, o valor que está armazenado na posição 3 do vetor (31) é atribuído à variável A.

É importante observar que a atribuição só é considerada válida porque P foi declarado como ponteiro para um *int*. Se P fosse declarado da seguinte forma

```
struct { int x; } *P;
```

A atribuição A = *P **não** seria válida e decorreria em erro na compilação, embora a aparência na memória fosse a mesma, já que a estrutura tem apenas um campo que é do tipo *int*.

No entanto, se P apontasse para uma estrutura, como proposto, como os tamanhos de ambas as variáveis são os mesmos, ainda parece fazer sentido a atribuição proposta. Com uma pequena modificação, proposta a seguir, a atribuição se torna possível.

```
A = *((int*)P)
```

Considerando que P aponta para uma estrutura, faz sentido que o programador deseje acessar os membros dessa estrutura.

Existem dois modos para acessar um campo de uma estrutura que é apontada por um ponteiro:

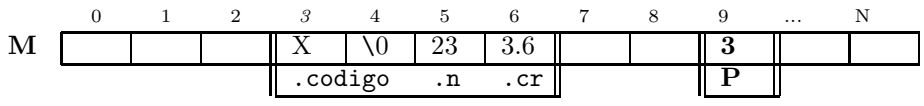
```
(*P).nomecampo;
P->nomecampo;
```

As declarações acima são equivalentes, ou seja, o resultado é o mesmo.

Tome como a exemplo a seguinte declaração:

```
struct {
  char codigo[2];
  int n;
  float cr;
} *P;
...
strcpy(P->codigo, "X");
P->n = 23;
(*P).cr = 13.6;
```

No vetor de memória a declaração poderia assumir o seguinte aspecto:



Por motivos de simplicidade, o exemplo acima assume que as variáveis do tipo *int* e *float* ocupam a mesma quantidade de espaço em memória, o que não acontece na realidade.

O processo de indexar a memória com o valor do ponteiro é também conhecido como *derreferenciamento* do ponteiro.

Até este momento, a utilização de ponteiros envolveu apenas indexar uma determinada região de memória, sem considerar a validade desta ação.

Na realidade, um ponteiro não pode ser efetivamente usado se não estiver apontando para um região válida da memória, ou seja, uma região que tenha sido alocada especificamente para o o seu programa.

No momento da declaração de um ponteiro, o seu conteúdo, que é a posição para onde está apontando, é *lixo*, como acontece na declaração de quaisquer outras variáveis. Portanto, é necessário especificar para onde deseja-se que o ponteiro aponte.

Existem dois modos de fazer um ponteiro apontar para uma posição válida:

- fazer o ponteiro apontar para uma variável existente no programa;
- alocar um espaço dinamicamente para o ponteiro.

Alocação dinâmica é assunto da próxima seção.

Para fazer um ponteiro apontar para uma variável existente no programa, deve haver um meio para obter o endereço da variável em questão.

O operador & , quando utilizado precedendo uma variável, obtêm o endereço de memória da variável em questão. Nesse contexto, o operador & é chamado de *referenciador*. É necessário cuidado para não confundir o operador de *referência* (que é unário) com o operador de *AND*(que é binário).

Dada a seguinte declaração de uma variável qualquer e a de um ponteiro para este tipo de variável:

```
tipo nome_var;
tipo* P;
```

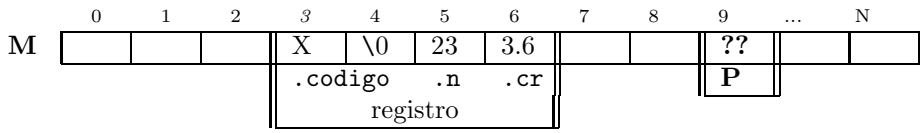
Para fazer como que o ponteiro P aponte para a variável *nome_var*, a seguinte declaração é necessária:

```
P = &nome_var;
```

Por exemplo, suponha as seguintes declarações:

```
struct {
  char codigo[2];
  int n;
  float cr;
} registro, *P;
```

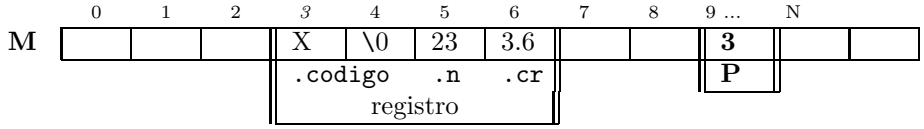
A memória assumiria o seguinte aspecto para a declaração acima:



Após a seguinte linha de código:

```
P = &registro;
```

A memória assumiria o seguinte aspecto:



Observe que o valor do ponteiro P agora é **3**, que é a posição de memória onde inicia a estrutura *registro*.

Observe o programa abaixo:

```
#include <stdio.h>

void main(){
  char c;
  char *pc;

  pc = &c; /* pc aponta para c */
  for (c = 'A'; c <= 'Z'; c++)
    printf (''%c'', *pc); /* derreferencia o ponteiro */
}
```

O programa acima imprime na tela o alfabeto em maiúsculas, nada muito misterioso. Mas considere *como* o programa cumpre esta simples tarefa. O programa acima realiza uma impressão que se refere apenas ao ponteiro *pc*.

A chave deste enigma está no ponteiro *pc*. Inicialmente, foi declarado um ponteiro para uma variável *char*. A seguir, é atribuída a posição de memória referente à variável *c* para o ponteiro *pc*.

Dessa forma, é possível acessar indiretamente o conteúdo da variável *c*, derreferenciando o ponteiro.

14.3.1 Exercício

Dadas as seguintes declarações:

```
struct computador {
  char processador[10]; /* nome do processador */
  int placa_video; /* codigo da placa de video */
  int modem:1; /* possui modem?  (s/n) */
  int modem_code:7; /* codigo do modem (se aplicavel) */
} comp_1;
typedef struct computador* comp_ponteiro;
```

Desenhe um diagrama da memória com a estrutura *comp_1* representada. Escreva o código necessário para que o ponteiro *comp_ponteiro* aponte para esta estrutura, e complete o diagrama com essa nova situação.

Declare um novo ponteiro *pcomp* que aponte para *comp_ponteiro* e escreva como deveria ser o código para acessar um dos campos da estrutura apontada por *comp_ponteiro* através do *pcomp*.

14.4 Passagem de parâmetros por referência

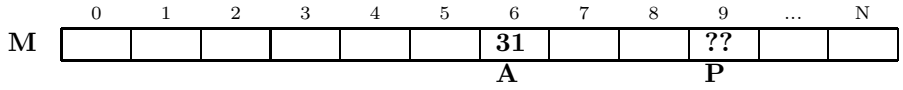
Muitas vezes, existe a necessidade de que funções alterem o valor de um ou mais parâmetros. Para que isso seja possível é utilizada uma técnica conhecida como *passagem de parâmetro por referência*, em contraposição ao método habitual que é a *passagem de parâmetro por valor*, a qual não reflete as alterações nas variáveis utilizadas na chamada da função.

Na linguagem C na verdade não existe nenhum mecanismo específico só para manipular a passagem de parâmetro por referência. Para que seja possível alterar o valor de uma variável passada como parâmetro é necessário se utilizar de ponteiros.

Em uma passagem por referência, é necessário que a variável de parâmetro seja, como o próprio nome da passagem sugere, uma referência para variável passada como parâmetro. Foi visto que ponteiros são capazes de apontar para variáveis e, portanto, podem servir como referência para uma variável.

Para entender como o processor funciona, analise novamente o diagrama do vetor que representa a memória.

```
int A, *P;
A = 31;
```



O diagrama acima apresenta a variável *A* e um ponteiro *P*. O objetivo é utilizar o ponteiro *P* para alterar o valor de *A*, ou seja, transformar *P* em uma referência para *A*.

Para tanto, basta examinar a seção 14.3.

A seguinte linha de código faz com que *P* aponte para *A*:

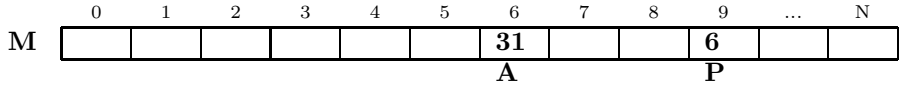
```
P = &A;
```

Desse modo, é possível alterar o valor de *A* através de *P*.

```
*P = 5;
```
⇒ *Equivalente a M[P]=5.*

Ao invés de declarar *P* como uma variável ordinária, é possível declarar uma função com o seguinte protótipo:

```
tipo_retorno func(int *P);
```



Dessa maneira, o que temos na verdade é uma função tal que um de seus parâmetros é um ponteiro.

Como já foi visto, para fazer com que este ponteiro aponte para uma variável qualquer, é necessário atribuir a ele o endereço da variável. Portanto, para que P seja realmente uma referência para a variável que será passada como parâmetro, a chamada da função *func* deverá ser da seguinte forma:

```
func(&A);
```

Ou seja, o que está acontecendo na realidade não é exatamente uma passagem por referência no sentido habitual do termo, mas a passagem do endereço de uma variável para que um ponteiro possa apontá-la e, por sua vez, através desse ponteiro a função será capaz de alterar o valor apontado pelo ponteiro.

14.4.1 Exercício

Escreva como seria o protótipo de uma função que receberá como um de seus parâmetros um ponteiro P, sendo que a função deverá ser capaz de alterar o ponteiro.

Escreva também qual seria a linha de código para alterar o valor do ponteiro, para que ele aponte para uma variável local da função A. Que problemas esse tipo atribuição poderia causar?

14.5 Aritmética de ponteiros

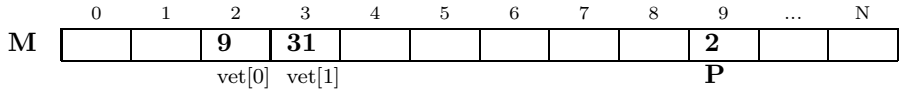
É interessante observar que, embora ponteiros sejam considerados um tipo especial de dados, eles ainda são números e, como tal, faz sentido realizar algumas operações matemáticas sobre eles.

A aritmética de ponteiros é restrita apenas a soma, subtração e comparações.

O que significa adicionar um número a um ponteiro? Para compreender qual o efeito de uma soma ou subtração sobre um ponteiro, é interessante recorrer novamente ao diagrama da memória.

Suponha as seguintes declarações:

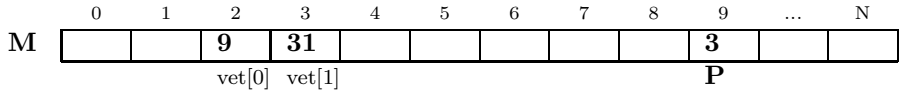
```
int vet[2] = {9, 31};
int *P;
p = &vet[0];  ⇒  faz P apontar para a 1ª célula de vet
```



Nesse caso, o conteúdo de $*P$ é 9. Se for realizada uma adição sobre P:

```
P++;
```

Teremos:



Agora, o conteúdo de *P é 31. Ou seja, realizar somas ou subtrações sobre ponteiros é nada mais do que fazer com que o ponteiro aponte para uma nova posição da memória.

Observe o seguinte exemplo:

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

int main (){
  char str[ ] = "Aritmetica de Ponteiros";
  char *p;
```

```
    p = &str[0];
    printf (``*p = %c\n'', *p);
    printf (``*(p+1) = %c\n'', *(p+1));
    printf (``*(p+5) = %c\n'', *(p+5));
    printf (``*(p+7) = %c\n'', *(p+7));
  }
```

A saída desse programa será:

```
*p     = A
*(p+1) = r
*(p+5) = é
*(p+7) = i
```

14.5.1 Exercício

Observe pelos exemplos, que a aritmética de ponteiros cria uma forte relação entre ponteiros e vetores. Somar valores a um ponteiro, faz o ponteiro apontar para uma próxima célula de um vetor.

Escreva um programa que obtenha como entrada do teclado uma palavra (que será armazenada em uma string) e imprima a string de trás para a frente.

14.6 Ponteiros e matrizes

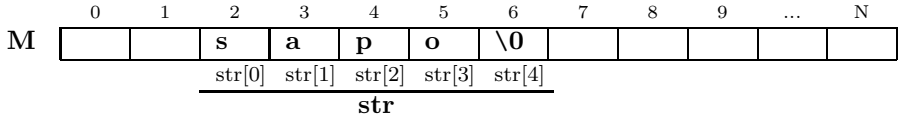
Mais do que apenas uma relação, para a linguagem C matrizes são, na realidade, ponteiros para a primeira posição da matriz na memória. É esta a razão pela qual matrizes sempre são passadas por referência para funções.

A única diferença entre uma matriz comum e um ponteiro, além da declaração, é que uma matriz é um tipo de dado cujo espaço é alocado no momento em que ela é declarada e seu ponteiro não pode ser alterado, assim como para qualquer variável não se pode alterar o seu próprio endereço, mas apenas seu conteúdo.

Suponha a seguinte declaração:

```
char str[5]="sapo";
```

A aparência dessa declaração na memória é a seguinte:

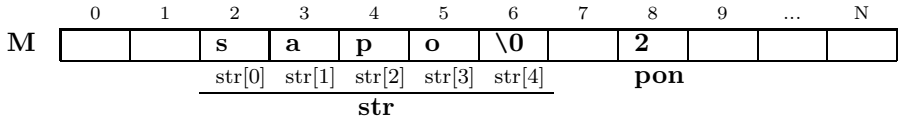


Note que a variável *str* sem indexação é na realidade um ponteiro para a posição 2 da memória.

Se a seguinte declaração é realizada:

```
char str[5]="sapo";
char *pon = str;
```

O efeito será:



A partir do momento que pon está apontando para o vetor *str*, este ponteiro poderá ser utilizado como se fosse o próprio vetor *str*.

Por exemplo:

`pon[3]` acessa o 3^o elemento de *str*(letra *o*).

Na linguagem C todo ponteiro pode ser indexado dessa maneira, mesmo que a posição indexada não "exista" (por exemplo, *pon[5]*). Naturalmente, uma tentativa de acessar posições não alocadas, quase sempre decorrem em erros (dependendo do sistema operacional, o erro não é muito aparente).

Na verdade, ao passar uma matriz para uma função, todo o processo acima ocorre naturalmente.

É interessante observar que para funções receberem matriz bidimensionais é necessário que seu parâmetro seja um ponteiro para ponteiro.

Por exemplo:

```
tipo_retorno func(int **matriz);
```

14.6.1 Exercício

Escreva uma função que receba duas matrizes 3x3, compute a soma dessas matrizes e imprima a resposta na tela.

14.7 Ponteiros para funções

Ponteiros para funções são tipos especiais de ponteiros. Em vez de armazenarem o endereço de uma área de dados, armazenam o endereço de uma função, ou seja, o endereço de memória para o qual o controle é transferido quando a função é chamada. O quadro a seguir mostra o modelo geral de declaração de ponteiros para funções e alguns exemplos:

```
<tipo-de-retorno> (*<nome-do-ponteiro>) (<parâmetros>);
void (*v_fptr) ();
int (*i_fptr) ();
double (*div_fptr) (int, int);
struct registro* (*reg_fptr) (char*, char*, int, char*);
```

Nos exemplos acima, *v_ptr* é um ponteiro para qualquer função que não recebe argumentos nem retorna nada. *reg_fptr* é um ponteiro para uma função que receba como parâmetro 3 strings e um inteiro, e retorne um ponteiro para uma struct registro.

Exemplo de utilização:

```
double f (double x)
{
  /* ...  */
}

double g (double x)
{
  /* ...  */
}
main ()
{
  /* ...  */
  double (*fp) (double);
  double x, y;

  /* ...  */
  if (x != 0) fp = f;
  else fp = g;

  /* ...  */
  y = (*fp) (x);

  /* ...  */
}
```

14.8 Problemas com ponteiros

Problemas com ponteiros são relativamente comuns e geralmente é muito de difícil de encontrar os erros relativos ao ponteiro.

O erro mais comum é a tentativa de usar um ponteiro que não esteja apontado para um posição válida de memória. Dependendo do sistema operacional, o problema não é imediatamente acusado, mas tende a ter dimensões catastróficas.

Imagine, por exemplo, que a posição para a qual o ponteiro aponta é uma região da memória utilizada pelo sistema operacional. Se o sistema operacional permitir a escrita nesta posição, o sistema pode até mesmo travar.

Exemplo:

```
void main(void){
  int x, *p;
  x = 10;
  (*p) = x; /* Em que endereço estamos armazenando x?  */
}
```

Atribui o valor 10 a alguma localização desconhecida da memória.

Capítulo 15

Alocação dinâmica de memória

Toda e qualquer informação que um programa utiliza está localizada na memória. Mas para que um programa possa utilizar uma área de memória para armazenar informação, é necessário que tal área seja previamente alocada, ou seja, é necessário requisitar ao sistema operacional que reserve uma área de memória para o programa e que a proteja, afim de que outros programas não venham a ler/gravar dados na região de memória reservada ao programa em questão.

Imagine se um programa que utilizasse, para armazenar um índice de um *for*, a mesma área de memória que outro programa usaria para armazenar uma entrada do teclado. Ou então, que a mesma área de memória venha a ser utilizada tanto para armazenar dados de um programa quanto para armazenar o código de outro programa em execução. Catástrofes de todos os tipos podem ocorrer em tais circunstâncias e se não houver um gerenciamento de memória por parte do sistema operacional, programar seria um desafio ainda maior, senão inviável.

Alocar uma área de memória significa pedir ao sistema operacional que reserve uma área para uso exclusivo do nosso programa.

15.1 Alocação estática × alocação dinâmica

Existem duas maneiras de se alocar memória em um programa em C.

A primeira maneira é chamada *alocação estática*. Quando o sistema operacional inicia a execução de um programa, ele aloca três regiões de memória: o segmento de código, o segmento de dados e o segmento de pilha.

No segmento de código o sistema operacional coloca o código do programa. No segmento de dados são colocadas as variáveis globais, constantes e variáveis static. No segmento de pilha são armazenadas, entre outras coisas, as variáveis locais das funções do programa.

O problema é que o tamanho desses segmentos é fixo (calculado pelo compilador), ou seja, não pode ser mudado durante a execução de um programa. Imagine que um programa, no meio de uma tarefa, necessite ler um arquivo de 2 Mb do disco, processá-lo e devolvê-lo para o disco. Se não for declarado no código do programa um array de 2 Mb de tamanho, não haverá como processar o arquivo.

Agora suponha que foi declarado um matriz de 2 Mb e o programa consegue manipular o arquivo. Suponha que esse é um arquivo de configuração e só precisa ser utilizado uma vez durante as 10 horas em que o programa ficou em execução. Como o tamanho dos segmentos é fixo (daí o nome *alocação estática*), esses 2 Mb de memória alocados estaticamente estariam reservados para o programa, mas não seriam utilizados (um exemplo de programa mal-comportado).

Ou seja, o programa estaria retendo 2 Mb de memória que poderiam ser usados por outros programas e essa memória alocada, mas não utilizada, pode trazer problemas, como impedir que se possa executar outros programas por falta de memória.

Poderia ser argumentado que pelo menos o programa funciona. Agora suponha que aquele arquivo tivesse seu tamanho aumentado para 2.5 Mb. Seria necessáro alterar no código o tamanho da matriz e recompilar o programa cada vez que mudasse o tamanho do arquivo.

O ideal é que esses 2 Mb de memória sejam alocados somente quando forem necessários e sejam liberados para outros programas quando deixassem de ser úteis.

É aí que entra a *alocação dinâmica*: a *alocação dinâmica* permite que o programa reserve uma área de memória de qualquer tamanho (dentro dos limites do tamanho da memória, é claro) em **tempo de execução**. Isso quer dizer que o programa/programador/compilador não precisa saber antecipadamente o tamanho do bloco de memória de que o nosso programa precisa.

Durante a execução, o programa descobre qual é o tamanho da área de memória que necessita e pede ao sistema operacional para reservar uma área de memória daquele tamanho. O sistema operacional reserva a área requisitada (se possível) e devolve para o programa o endereço do primeiro byte da área de memória alocada. No programa, esse endereço pode ser armazenado em um ponteiro.

15.2 *sizeof*

Antes de apresentar as funções de manipulação de memória dinâmica, é importante descrever o operador *sizeof*.

O operador *sizeof* é usado para obter o tamanho, em bytes, de um determinado tipo de dado.

A sintaxe geral é:

`sizeof(tipo)` ou ainda `sizeof(`*`variável`*`)`

O *sizeof* retorna o tamanho do *tipo* passado como parâmetro ou do tipo da *variável* passada como parâmetro.

Exemplos:

```
struct coord{
  int x, y, z;
};
struct coord coordenada1;
sizeof(struct coord);  ⇒  obtêm o valor 12 (4 bytes por int)
sizeof(coordenada1);  ⇒  obtêm o valor 12 (4 bytes por int)
sizeof(int);  ⇒  obtêm o valor 4
```

Esse operador é extremamente útil quando é necessário trabalhar com alocação dinâmica de memória porque permite ao programa determinar o quanto de memória deve ser alocado para um determinado tipo de dado.

15.3 Função *malloc()*

A função *malloc* requisita ao sistema operacional para alocar uma área de memória do tamanho especificado.

Essa função é extremamente útil para gerar matrizes cujo tamanho não é possível ser definido antes de executar o programa. Além disso, existem estruturas de dados (listas, filas, pilhas, entre outras) que tem tamanho variável e precisam dessa função para serem implementadas.

O protótipo da função *malloc* é:

```
void *malloc (unsigned int numero_de_bytes);
```

A função recebe como argumento o tamanho em bytes de memória que se deseja alocar e devolve um ponteiro do tipo *void** para o primeiro byte da área de memória alocada.

Em caso de erro (não há memória suficiente), o valor retornado é **NULL**.Como um ponteiro do tipo *void** não tem tipo definido, pode ser utilizado um casting para especificar que tipo de ponteiro ele deverá ser.

Em algumas implementações da linguagem C, o compilador limita a quantidade de memória que o programador pode alocar (ex.: no compilador **Borland C 3.0**, o máximo é 64Kb).

Exemplo:

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

int main(){
  int* vetor, tamanho;
  printf("Digite o tamanho do vetor:");
```

```
  scanf("%d", &tamanho);
  vetor = (int*)malloc(sizeof(int)*tamanho);
}
```

Observe que para alocar o tamanho correto para um vetor de *int*, é necessário multiplicar o número de células que se deseja pelo tamanho do tipo de dado *int* porque cada célula individual é do tamanho *int*.

15.4 Função *free()*

A função *free* é a inversa da função *malloc*, isto é, ela desaloca (libera) uma área de memória previamente alocada pela função malloc. Abaixo temos a declaração da função:

```
void free (void *memblock);
```

A função recebe um único argumento, o qual é um ponteiro para uma área de memória previamente alocada com *malloc*.

```
/* calcula a media de n notas especificadas pelo usuario */
#include <stdlib.h>
#include <stdio.h>

int main(){
  double *notas, media=0;
  int i, n;

  printf ("Digite o numero de notas:  ");
  scanf ("%d", &n); /* obtem o numero de notas */

  if (n < 1){ /* verifica se o numero e valido */
    printf ("Numero invalido!  Deve ser maior que zero.\n");
    return;
  }
  /* aloca memoria */
  notas = (double*) malloc (n * sizeof(double));

  for (i=0; i<n; i++){ /* obtem as notas */
    printf ("Digite a %da.  nota:  ", i+1);
    scanf ("%f", (notas+i));
  }
  /* calcula a media das notas */
  for (i=0; i<n; i++)
    media += *(notas+i);
  media /= n;

  printf ("A media das notas e:  %f\n", media);
  free (notas); /* desaloca a memoria alocada previamente */
  }
```

15.5 Exercícios

1) Faça uma função *gera_matriz* com os seguintes parâmetros:

```
geramatriz(int x, int y, int z, int tam);
```

A função deverá alocar uma matriz de dimensão 3, cada dimensão deverá ter tamanho *tam* e, ao final, a função retorna a matriz.

2) Crie uma função que seja capaz de redimensionar um vetor previamente alocado (vetor de dimensão 1).

A função será chamada realoca e receberá os seguintes parâmetros:

```
realoca(int *vetor, int tam, int novo_tam);
```

Onde vetor é o vetor que deve ser realocado, *tam* é o tamanho velho do vetor e *novo_tam* é o novo tamanho.

A função deverá ser capaz de preservar o conteúdo de vetor. Se o *novo_tam* for menor que *tam*, a informação das células excedentes deve ser descartada.

Capítulo 16

Arquivos

Como não podia deixar de ser, a linguagem C possui uma série de funções que permitem que o programador possa manipular arquivos, seja para criá-los, ler ou escrever neles.

As funções descritas nesse capítulo utilizam ponteiros do tipo *FILE** para manipular arquivos, seja escrita ou leitura.

Declaração de uma variável ponteiro de arquivo:

```
FILE *fp;
```

16.1 Funções para manipulação de arquivos

Função	Operação
`fopen()`	Abre um arquivo
`fclose()`	Fecha um arquivo
`fputc()`	Escreve um caractere em um arquivo
`fgetc()`	Lê um caractere de um arquivo
`fseek()`	Procura por uma posição do arquivo
`fprintf()`	Grava uma string num arquivo, com formatação (saída formatada)
`fscanf()`	Lê um string do arquivo, com formatação (entrada formatada)
`feof()`	Retorna verdadeiro se o fim do arquivo é encontrado
`ferror()`	Retorna verdadeiro se ocorreu um erro
`fread()`	Lê um bloco de dados de um arquivo
`fwrite()`	Escreve um bloco de dados em um arquivo
`rewind()`	Reposiciona o ponteiro de arquivo no começo do arquivo
`remove()`	Apaga o arquivo

Tabela 16.1: Funções mais usadas

Veja a lista das funções mais comuns na tabela 16.1. Todas as funções apresentadas estão na biblioteca `stdio.h`.

16.2 *EOF*

EOF é uma constante definida na biblioteca *stdio.h* que é utilizada para indicar o fim de um arquivo. No geral, qualquer função que realiza leitura sobre um arquivo retorna *EOF* se não houver mais nenhum caracter a ser lido, ou seja, caso tenha sido atingido o final do arquivo.

16.3 Função *fopen()*

Protótipo:

```
(FILE *)fopen(char *nome_do_arquivo, char *modo);
```

Esta função abre um arquivo e retorna um ponteiro de arquivo. O primeiro argumento é o nome do arquivo. O segundo é uma string de formatos, listados na tabela 16.2.

Modo	Significado
"r"	Abre um arquivo para leitura
"w"	Cria um arquivo para escrita
"a"	Acrescenta dados para um arquivo existente
"rb"	Abre um arquivo binário para leitura
"wb"	Cria um arquivo binário para escrita
"ab"	Acrescenta dados a um arquivo binário já existente
"r+"	Abre um arquivo para leitura/escrita
"w+"	Cria um arquivo para leitura/escrita
"a+"	Acrescenta dados ou cria um arquivo para leitura/escrita
"r+b"	Abre um arquivo binário para leitura/escrita
"w+b"	Cria um arquivo binário para leitura/escrita
"a+b"	Acrescenta ou cria um arquivo binário para leitura/escrita
"rt"	Abre um arquivo texto para leitura
"wt"	Cria um arquivo texto para leitura
"at"	Acrescenta dados a um arquivo texto
"r+t"	Abre um arquivo texto para leitura/escrita
"w+t"	Cria um arquivo texto para leitura/escrita
"a+t"	Acrescenta dados ou cria um arquivo texto para leitura/escrita

Tabela 16.2: Modos de acesso a arquivo

Exemplo:

```
FILE *fp; /* fp e um ponteiro para arquivo */
if ((fp=fopen("teste.txt", "w")) == NULL) {
  /* se não conseguiu criar arquivo ...  */
  puts ("Nao posso criar o arquivo!\n");
  exit(1);
}
```

Observação: Ao tentar abrir um arquivo já existente com a opção "w", será criado um novo arquivo em disco, apagando o antigo.

16.4 Função *fclose()*

A função *fclose* é usada para fechar um arquivo que foi aberto por *fopen*. Ela escreve quaisquer dados restantes do buffer para o disco e faz um fechamento formal em nível de sistema operacional.

Protótipo:

```
int fclose (FILE *fp);
```

Onde *fp* é um ponteiro para um arquivo . Um valor de retorno igual a zero significa que a operação foi realizada com sucesso, qualquer outro valor significa erro. Geralmente o único momento em que a função fclose falhará é quando um disquete tiver sido removido do drive.

16.5 Função *fputc()*

A função *fputc* é usada para escrever caracteres em um arquivo aberto para escrita.

Protótipo:

```
int fputc(int ch, FILE *fp);
```

Onde *ch* é o caractere a ser escrito e *fp* é um ponteiro para um arquivo. Se não houver erro a função retornará o caractere escrito. Em caso de falha, um *EOF* é retornado.

16.6 Função *fgetc()*

A função *fgetc()* é usada para ler caracteres de um arquivo.

Protótipo:

```
int getc(FILE *fp);
```

Onde *fp* é um ponteiro para um arquivo. A função retorna o caracter lido.

Exemplo:

```
/* O programa que le arquivos e exibe-os na tela */
#include <stdio.h>
#include <stdlib.h>

void main (int argc, char *argv[ ]) {
  FILE *fp;
  char ch;
  if (argc !=2 ){
    printf ("Voce esqueceu de informar o nome do arquivo!\n");
    exit (1);
  }
  if ((fp=fopen(argv[1], "r"))==NULL){
    printf("O arquivo não pode ser aberto\n");
    exit (1);
  }
  ch = getc(fp) /* lê um caractere */
  while (ch != EOF){
    /* repete enquanto nao for o final do arquivo */
    printf("%c", ch); /* imprime caracter lido na tela */
    ch = getc(fp); /* le proximo caracter */
  }
  fclose(fp); /* fecha o arquivo */
}
```

16.6.1 Exercícios

1) Escreva um programa que leia palavras digitadas pelo usuário enquanto ele não digitar "fim"e que escreva estas strings em um arquivo chamado strings.dat.

O programa deverá gravar o final de cada string, ou seja, o caracter '\0'.

2) Escreva um programa que abra o arquivo strings.dat criado pelo programa do exercício anterior e imprima as strings armazenadas na tela.

A cada vez que for encontrado um final de string, o programa deverá quebrar a linha (imprimir o caracter '\n').

16.7 Função *feof()*

Quando um arquivo é aberto para entrada binária, é possível encontrar um valor inteiro igual à marca de *EOF*. Isso pode fazer com que seja indicada uma condição de fim de arquivo, sem que o mesmo tenha sido realmente encontrado.

Para resolver este problema, incluiu-se a função *feof()*, que é usada para determinar o final de um arquivo quando da leitura de dados binários.

Protótipo:

```
int feof(FILE *fp);
```

Onde *fp* é um ponteiro de um arquivo usado por *fopen()*. O valor de retorno é verdadeiro (1) se o fim do arquivo foi encontrado.

Exemplo:

```
while (!feof(fp)) ch=getc(fp);
```

Naturalmente, o mesmo método pode ser aplicado tanto a arquivos textos como a arquivos binários.

16.8 Função *ferror()*

A função *ferror* é usada para determinar se uma operação em um arquivo produziu erro.

Protótipo:

```
int ferror(FILE *fp);
```

Onde *fp* é um ponteiro para um arquivo. A função *ferror* retorna verdadeiro (1) se um erro ocorreu durante a última operação com o arquivo e falso (0), caso contrário.

Uma vez que cada operação em arquivo determina uma condição de erro, a função ferror deve ser chamada imediatamente após cada operação com o arquivo, caso contrário, um erro pode ser perdido.

16.9 Função *rewind()*

A função *rewind* reinicia o ponteiro do arquivo para o começo do mesmo.

Protótipo:

```
void rewind(FILE *fp);
```

Onde *fp* é um ponteiro de arquivo.

16.10 Função *remove()*

A função *remove()* apaga o arquivo especificado.

Protótipo:

```
int remove(char *nome_do_arquivo);
```

Onde *nome_do_arquivo* é uma string contendo o nome do arquivo e o valor de retorno é 0 em caso de sucesso e diferente de zero se ocorrer um erro.

16.11 Funções *fgets()* e *fputs()*

Essas duas funções podem ler e escrever strings para fluxos.

Protótipos:

```
char *fputs(char *str, FILE *fp);
char *fgets(char *str, int tamanho, FILE *fp);
```

A função *fputs* escreve a string em um arquivo *fp* especificado.

A função *fgets* lê uma string do arquivo *fp* especificado. A função fgets lê uma string até que um caractere de nova linha seja lido ou *tamanho*-1 caracteres sejam lidos. Se uma nova linha é lida, o caracter nova linha será adicionado ao final da string *str*.

Toda string lida por fgets é terminada pelo caracter nulo `\0`.

16.11.1 Exercícios

Resolva os dois exercícios da seção 16.6.1, na página 61, mas agora utilizando as funções *fgets* e *fputs* ao invés de *getc* e *putc*.

Nesse caso, não se preocupe em gravar o caracter nulo.

16.12 Funções *fread()* e *fwrite()*

As funções *fread* e *fwrite* permitem leitura ou escrita de um ou mais blocos de dados ao mesmo tempo. Estas funções são muito úteis para gravar diversos tipos de dados mais complexos, como vetores e estruturas.

Protótipos:

```
unsigned fread(void *buf, int num_bytes, int count, FILE *fp);
unsigned fwrite(void *buf, int num_bytes, int count, FILE *fp);
```

Na função *fread()*, *buf* é um ponteiro ou uma referência para uma região de memória que receberá os dados lidos do arquivo.

Observe que para obter uma referência para uma determinada operação, pode ser utilizado o operador & precedendo variável (ver seção 14.3, p.47). *buf* pode ser referência para qualquer tipo de dado.

O parâmetro *num_bytes* é o tamanho, em *bytes*, do dado passado como parâmetro para leitura em *buf*.

Como já foi visto na seção 15.2, p.56, o operador *sizeof* é ideal para obtenção do tamanho de um dado.

O parâmetro *count* indica quantos dados do tamanho *num_bytes* deverão ser lidos para a memória, ou seja, *count* é ideal para a leitura de vetores a partir de uma arquivo.

Por final, o parâmetro *fp* é um ponteiro para um arquivo a partir do qual os dados serão lidos.

Exemplo:

```
#include <stdio.h>

int main(){
  struct { int x, y, z; }, est, mat[5];
  /* abre arquivo para leitura, modo binario */
  FILE *fp = fopen("arq", "rb");

  fread(&est, sizeof(est), 1, fp); /* le dados para variavel est */
  fread(&mat, sizeof(est), 5, fp); /* le dados para matriz mat */
  fclose(fp);
}
```

Note, no exemplo anterior, que para realizar uma leitura sobre o vetor *mat* não foi necessário preceder o nome da variável com o operador &. Isso é possível porque toda matriz em C é considerada como se fosse um ponteiro (uma referência para o início da matriz), como já foi visto na seção 14.6, p.52.

Para a função *fwrite*, os parâmetros têm significado semelhante aos já descritos.

buf é um ponteiro(referência) para um dado que deseja-se escrever.

num_bytes é o tamanho do tipo de dado.

count é a quantidade de dados de tamanho *num_bytes* que deve ser escrito.

fp é um ponteiro para o arquivo em que deverá ser realizada a escrita.

Exemplo

```
#include <stdio.h>
#include <stdlib.h>
int main(){
  FILE *fp;
  float exemplo[100];
  int i;

  /* verifica se houve erro na abertura do arquivo */
  if ((fp=fopen ("exemplo", "wb")) == NULL) {
    printf ("O arquivo não pode ser aberto!\n");
    exit(1);
  }

  /* le a matriz inteira em um unico passo */
  if (fread(&exemplo, sizeof(exemplo), 1, fp) != 1)
    printf("Erro no arquivo!");
      for (i=0; i<100; i++) printf ("%f", exemplo[i]);
      fclose(fp);
    }
```

16.12.1 Exercícios

1) Escreva um programa que armazena dados digitados pelo usuário em uma estrutura *cliente* e grava esta estrutura em um arquivo enquanto o campo código não for 0.

```
struct cliente{
  int codigo;
  char nome [20];
};
```

Os primeiros bytes desse arquivo deverão ser reservados para gravar o número de registro gravados. A variável de contagem deverá ser do tipo *long int.*

2) Escreva uma programa que abre o arquivo gerado pelo programa do exercício anterior, lê os primeiros bytes para uma variável e finalmente lê todos os dados do arquivo de uma vez para um vetor clientes.

Note que o tamanho do vetor *clientes* deverá ser determinado dinamicamente, a partir da variável que diz quantos clientes foram previamente gravados.

16.13 Funções *fprintf()* e *fscanf()*

As funções *fprintf* e *fscanf* funcionam exatamente como as funções *printf* e *scanf*, mas ao invés de operarem sobre a entrada e saída padrão, operam sobre um arquivo qualquer.

Protótipos:

```
int fprintf(FILE *fp, char *string_de_controle,....);
int fscanf(FILE *fp, char *string_de_controle,....);
```

Onde *fp* é um ponteiro para um arquivo no qual deseja-se ler ou escrever.

16.13.1 Exercício

Apresente um código utilizando *fprintf* e *fscanf* ao invés de *fwrite* e *fread* que seja equivalente ao código abaixo:

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

int main(){
  struct cliente {
    int codigo;
    char nome [30];
  };
  struct cliente cli_1 = { 20, "Fulano Ciclano de Beltrano"};
  struct cliente cli_2;
  FILE* fp = fopen("arquivo", "wb");
  fwrite(&cli_1, sizeof(struct cliente), 1, fp);
  fclose(fp);
  fp = fopen("arquivo", "rb");
  fread(&cli_2, sizeof(struct cliente), 1, fp);
  printf("Nome %s codigo %d\n", cli_2.nome, cli_2.codigo);
}
```

16.14 Função *fseek()*

Através da função *fseek* é possível posicionar o ponteiro de leitura/escrita de um arquivo para qualquer posição dentro do arquivo.

Protótipo:

```
int fseek (FILE *fp, long numbytes, int origem);
```

Origem	Nome da Constante	Valor
Começo do arquivo	SEEK_SET	0
Posição corrente	SEEK_CUR	1
Fim do arquivo	SEEK_END	2

Tabela 16.3: Constantes de posição

Onde *fp* é um ponteiro para um arquivo, *numbytes*, é o número de bytes, a partir da origem, necessários para se conseguir a posição corrente e origem é uma das constantes definidas em *stdio.h* listadas na tabela 16.3.

Observe que, através da função *fseek*, é possível posicionar a leitura/escrita para o início do arquivo (`fseek(fp, 0, SEEK_SET)`), tornando-a equivalente à função *rewind*.

Se a função *fseek* retornar zero, indica que houve sucesso na chamada à função *fseek*. Um valor diferente de zero indica uma falha.

```
/* ...  */
FILE *fp;
char ch;
if ((fp=fopen("teste", "rb")) == NULL){
  printf ("o arquivo não pode ser aberto\n");
  exit (1);
}
fseek(fp, 234, SEEK_SET); /* pode-se trocar o SEEK_SET por 0 */
ch = getc(fp); /* lê o 235º caracter*/
/* ...  */
```

16.15 Exercícios

1. Escreva um programa que receba dados informados pelo usuário em uma estrutura *questao*, definida abaixo:

   ```
   struct questao{
     char enunciado[255];
     int n_opcoes, soma_correta;
     char** texto_opcao;
   }
   ```

   O funcionamento do programa deverá ser o seguinte:

   - recebe-se o enunciado da questão;
   - recebe-se o número de opções desta questão;
   - recebe-se o texto referente a cada opção, o qual é armazenado na matriz *texto_opcao*;
   - a cada vez que o programa requisita que o usuário digite o texto, deverá ser apresentado o número de soma da questão (que deverá ser sempre uma potência de 2).

   A matriz *texto_opcao* é uma matriz bidimensional, que será indexada por 0..n para determinar o número da opção, e para cada opção deverão ser alocados 128 bytes para entrar com a string.

   Por final, o programa deve requisitar que seja digitado o valor da soma correta.

   Depois de terminada a entrada de dados, o programa deverá armazenar a estrutura em um arquivo chamado *"questoes.dat"* e perguntará ao usuário se deseja gravar mais questões.

2. Escreva um programa que vai ler o arquivo *"questoes.dat"* criado pelo programa do exercício anterior, e simplesmente irá imprimir o enunciado da questão, o texto das opções com seus respectivos números e o valor da resposta.

Apêndice A

Palavras-Chave ou reservadas

As seguintes palavras são reservadas e não podem ser definidos símbolos nomeados segundo as strings abaixo:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Apêndice B

Bibliotecas

Para inserir uma biblioteca em um programa C, basicamente são necessários 2 passos:

- inserir o header no código fonte;
- na compilação, se necessário, deverá ser especificado o arquivo objeto;

No código fonte, a inserção do header sempre segue o seguinte formato:

```
#include <nomelib>
```

ou ainda

```
# include "nomelib"
```

Sendo que no primeiro formato, o compilador supõe que o arquivo *nomelib* se encontra em um diretório padrão para bibliotecas.

No segundo formato, o compilador supõe que o arquivo *nomelib* se encontra no diretório corrente.

Nos dois formatos é possível especificar explicitamente em que diretório se encontra o arquivo de header.

Ex.: **#include </home/user/biblio.h>**

Os headers das bibliotecas mais comuns são:

stdlib.h - contém as funções padrões do C;

stdio.h - contém as funções padrões de entrada e saída do C;

string.h - contém as funções de manipulação de strings;

math.h - contém funções matemáticas.

O segundo passo, referente à especificação dos arquivos objetos da biblioteca, muda de acordo com o compilador.

B.1 Bibliotecas: Arquivo objeto × Arquivo header

Uma distinção importante quanto a bibliotecas é entre arquivos objeto e "headers"(cabeçalhos).

Um arquivo objeto é um programa quase inteiramente em código de máquina (linguagem que o computador entende) a não ser por algumas referências internas tais como, por exemplo, chamadas de função.

O arquivo header (<bibname>**.h**) é o arquivo que apenas descreve os nomes das funções daquela biblioteca, além de declarar algumas variáveis, constantes, inclusão de outros arquivos, etc. A implementação das funções da biblioteca geralmente não está no header, e sim no arquivo objeto da biblioteca.

Ou seja, o arquivo objeto de uma biblioteca é onde se encontra realmente o código de implementação de todas as funções da biblioteca.

B.2 Bibliotecas: Lista de funções

Neste apêndice apresentamos apenas as funções que são usadas direta ou indiretamente neste curso, seja em exemplos ou em exercícios.

B.2.1 Funções padrão (*stdlib.h*)

Funções:

abort	abs	atexit	atof
atoi	atol	bsearch	calloc
div	ecvt	exit	_exit
fcvt	free	_fullpath	gcvt
getenv	itoa	labs	ldiv
lfind	_lrotl	_lrotr	lsearch
ltoa	_makepath	malloc	max
mblen	mbtowc	mbstowcs	min
putenv	qsort	rand	random
randomize	realloc	_rotl	_rotr
_searchenv	_splitpath	srand	trtod
strtol	_strtold	strtoul	swab
system	time	ultoa	wctomb
wcstombs			

Constantes, tipos de dados e variáveis globais:

div_t	_doserrno	environ	errno
EXIT_FAILURE	EXIT_SUCCESS	_fmode	ldiv_t
NULL	_osmajor	_osminor	_psp
RAND_MAX	size_t	sys_errlist	sys_nerr
_version	wchar_t		

B.3 Funções de entrada e saída padrão *stdio.h*

Funções:

clearerr	fclose	fcloseall	fdopen	feof	ferror
fflush	fgetc	fgetchar	fgetpos	fgets	fileno
flushall	fopen	fprintf	fputc	fputchar	fputs
fread	freopen	fscanf	fseek	fsetpos	ftell
fwrite	getc	getchar	gets	getw	perror
printf	putc	putchar	puts	putw	remove
rename	rewind	rmtmp	scanf	setbuf	setvbuf
sprintf	sscanf	strerror	_strerror	tempnam	tmpfile
tmpnam	ungetc	unlink	vfprintf	vfscanf	vprintf
vscanf	vsprintf	vsscanf			

Constantes, tipos de dados e variáveis globais:

buffering modes	BUFSIZ	EOF
_F_BIN	_F_BUF	_F_EOF
_F_ERR	_F_IN	_F_LBUF
_F_OUT	_F_RDWR	_F_READ
_F_TERM	_F_WRIT	FILE
FOPEN_MAX	fpos_t	fseek/lseek modes
_IOFBF	_IOLBF	_IONBF
L_ctermid	L_tmpnam	NULL
SEEK_CUR	SEEK_END	SEEK_SET
size_t	stdaux	stderr
stdin	stdout	stdprn
SYS_OPEN	TMP_MAX	

B.3.1 Funções de manipulação de strings (string.h)

Funções:

_fmemccpy	_fmemchr	_fmemcmp	_fmemcpy	_fmemicmp
_fmemset	_fstrcat	_fstrchr	_fstrcmp	_fstrcpy
_fstrcspn	_fstrdup	_fstricmp	_fstrlen	_fstrlwr
_fstrncat	_fstrncmp	_fstrnicmp	_fstrncpy	_fstrnset
_fstrpbrk	_fstrrchr	_fstrrev	_fstrset	_fstrspn
_fstrstr	_fstrtok	_fstrupr	memccpy	memchr
memcmp	memcpy	memicmp	memmove	memset
movedata	movmem	setmem	stpcpy	strcat
strchr	strcmp	strcmpi	strcpy	strcspn
strdup	_strerror	strerror	stricmp	strlen
strlwr	strncat	strncmp	strncmpi	strncpy
strnicmp	strnset	strpbrk	strrchr	strrev
strset	strspn	strstr	strtok	strxfrm
strupr				

Constantes, tipos de dados e variáveis globais:

size_t

B.3.2 Funções matemáticas (*math.h*)

Funções:

abs	...	acos	acosl	asin	asinl
atan	atanl	atan2	atan2l	atof	_atold
cabs	cabsl	ceil	ceill	cos	cosl
cosh	coshl	exp	expl	fabs	fabsl
floor	floorl	fmod	fmodl	frexp	frexpl
hypot	hypotl	labs	...	ldexp	ldexpl
log	logl	log10	log101	matherr	_matherrl
modf	modfl	poly	polyl	pow	powl
pow10	pow10l	sin	sinl	sinh	sinhl
sqrt	sqrtl	tan	tanl	tanh	tanhl

Constantes, tipos de dados e variáveis globais:

complex (struct)	_complexl (struct)	EDOM
ERANGE	exception (struct)	_exceptionl (struct)
HUGE_VAL	M_E	M_LOG2E
M_LOG10E	M_LN2	M_LN10
M_PI	M_PI_2	M_PI_4
M_1_PI	M_2_PI	M_1_SQRTPI
M_2_SQRTPI	M_SQRT2	M_SQRT_2
_mexcep		

Apêndice C

GCC - Compilação em Linux

O gcc é um compilador C padrão para Linux, desenvolvido pela GNU, um grupo de desenvolvimento de software para Linux. Sua sintaxe é:

```
gcc [opções] arquivo [arq1 arq2 ...]
```

Onde *arquivo* é o arquivo fonte ou objeto. Além do primeiro arquivo, é possível relacionar outros arquivos que deverão ser compilados e reunidos em um só arquivo executável.

As opções são parâmetros facultativos que alteram o comportamento do gcc. Segue abaixo uma lista da opções mais comuns:

- **-c** : apenas compila e gera um arquivo objeto (nome de saída padrão é arquivo.o);
- **-o <nomearq>** : especifica que o nome do arquivo de saída será *nomearq*;
- **-g** : gera informação de "debug", usado por programas de depuração (ex.: gdb);
- **-l<lib>** : especifica o arquivo objeto das bibliotecas que deverão ser incluídas no processo de compilação.

É importante observar que a maioria das bibliotecas padrão não precisam ser especificadas através da opção -l (ex.: stdlib, stdio, string, etc).

Em dúvida, uma consulta do **man** sobre o comando de uma determinada biblioteca pode ajudar a encontrar a biblioteca que deve ser incluída.

Todas as bibliotecas C em linux iniciam com o nome **lib**, que não deve ser usado na especificação por -l, ou seja, apenas a string remanescente (sem lib) deve ser usada.

Por exemplo, o arquivo objeto da biblioteca de funções matemáticas, cujo header é *math.h* chama-se **libm**. Então, para compilar um programa que inclui a *math.h* basta inserir a opção **-lm** para incluir a libm. No linux, geralmente as bibliotecas estão no diretório /usr/lib.

Exemplo:

Suponha que você tenha uma arquivo fonte de nome matriz.c que utiliza a biblioteca math.c, que não é padrão, e quer gerar um executável de nome matriz. A linha de comando que realiza exatamente o desejado é:

```
gcc -o matriz -lm matriz.c
```

Onde *-o matriz* especifica o arquivo de saída, *-lm* especifica que a biblioteca libm deverá ser incluída e *matriz.c* é o nome do arquivo fonte.

Apêndice D

Módulos

D.1 Modulando programas em C

Uma das características mais interessantes da linguagem C é a possibilidade de dividir um programa em vários arquivos diferentes (módulos).

Modular um programa em C é relativamente simples. Os passos necessários são:

- criar vários arquivos que vão possuir o código dos módulos;
- criar arquivos cabeçalhos (.h);
- gerar os arquivos objetos e linkar todos os programas juntos.

Nessa seção, vamos falar especificamente de compilação e modulação em Linux, mas os passos acima são válidos em qualquer sistema operacional, embora sejam realizados de maneira diferente.

Para ilustrar o processo, vamos desenvolver um programa e um módulo que será utilizado por este programa.

```
/* modulo imp_fat.c */
#include <stdio.h>
#include "fatorial.h"
int main(){
  long int i, j;
  printf("Digite o numero do qual voce deseja obter o fatorial:");
  scanf("%ld", &i);
  j = fatorial(i);
  printf("O fatorial de %ld e %ld", i, j);
}
/* cabecalho fatorial.h */
long int fatorial(long int n);

/* modulo fatorial.c */
#include <fatorial.h>
long int fatorial(long int n){
  if (n<2) return 1;
  return n*fatorial(n-1);
}
```

Um arquivo cabeçalho nada mais é do que um arquivo que possui todas as declarações de funções, variáveis globais e constantes de um módulo.

O arquivo *fatorial.h* é um arquivo cabeçalho que possui a declaração da função fatorial. Para um compilador C, essa informação é suficiente para criar o arquivo objeto, mas não para gerar o executável. Para gerar um arquivo executável é necessário que exista acesso à definição da função, que está no módulo *fatorial.c*.

Usando o compilador *gcc*, para poder juntar os módulos acima e gerar um executável, basta digitar a seguinte linha de comando:

```
gcc -o nome_exec fatorial.c imp_fat.c
```

O comando anterior vai gerar um arquivo executável chamado *nome_exec*.

O método genérico para compilar vários módulos juntos é:

```
gcc -o nome_exec arq1 arq2 arq3 ..  arqn
```

Onde *arq1...arqn* podem ser tanto arquivos fontes (arquivo texto com código C) como arquivos objetos, gerados previamente pelo compilador gcc com a opção -c.

D.2 Make

O *make* é um utilitário de pré-processamento de arquivos, que permite realizar ações sobre determinados arquivos condicionadas à data de alteração de suas *dependências*.

As *dependências* de um arquivo são outros arquivos do qual ele depende. Por exemplo, um programa que tem função de manter um banco de dados atualizado a partir de um arquivo texto, tem este arquivo texto como sua dependência.

No que se refere à programação, o make pode ser utilizado para manter atualizado um programa que têm como dependência vários módulos.

O make utiliza normalmente um arquivo chamado *Makefile*(ou *makefile*) que descreve quais são as dependências de um programa e quais as ações que devem ser realizadas se alguma dependência estiver desatualizada ou simplesmente não existir.

É possível criar makefile's que compilam um determinado programa se os arquivos fontes e/ou objetos do qual ele depende não estiverem atualizados ou não existirem.

Em um makefile, basicamente os seguintes elementos podem ser utilizados:

- **dependências**: é uma linha que atribui um rótulo para uma dependência, listando quais são os arquivos que são dependências.

Forma:

```
nome_depend:  depend1 depend2 depend3
```

As dependências *depend1...dependn* devem ser arquivos. O nome_depend pode ser o nome de um arquivo ou um rótulo.

Para cada dependência *depend*, pode ser descrito quais são as dependências de depend, caso exista alguma.

- **ação**: é uma linha que descreve que ação deve ser realizada se as dependências não estiverem atualizadas

Forma:

```
qualquer programa executável
```

Exemplo:

```
Arquivo makefile:
nome_exec:  fatorial.o imp_fat.o gcc -o nome_exec fatorial.o imp_fat.o
fatorial.o:  fatorial.c fatorial.h
    gcc -c fatorial.h
imp_fat.o:  imp_fat.c fatorial.h
    gcc -c imp_fat.c
```

O makefile acima, foi criado para os módulos descritos nos exemplos da seção anterior.

Note que atualizar uma dependência, nesse caso, significa compilar os módulos necessários para gerar o executável *nome_exec*.

Para que as dependências sejam checadas e atualizadas, basta digitar *make* na linha de comando no diretório em que se encontram os arquivos fontes e o makefile.

Apêndice E

Recursividade

Recursividade é uma técnica de programação que envolve utilizar definições recursivas de modo a simplificar vários algoritmos.

Uma definição recursiva é uma definição que utiliza a si mesmo para se definir. A princípio, a idéia pode parecer confusa e obscura, mas na realidade é um conceito relativamente simples.

Por exemplo, é possível definir uma exponenciação dessa maneira:

Seja n, k $\in$ N,
$n^0 = 1$
$n^k = n.n^{k-1}$

Observe que, no exemplo acima, a exponenciação n_k está sendo definida através de uma outra exponenciação (n^{k-1}), ou seja, este é um caso em que a exponenciação é definida através dela mesma (o que é uma definição *recursiva* ou também chamada de *recorrência*).

Analisando um pouco melhor o exemplo acima, n^{k-1} também é uma exponenciação, portanto poderia utilizar a mesma definição para se definir, ou seja, se tomamos $n^{k-1} = n.n^{k-2}$ e assim podemos definir n^{k-2} , $nk - 3$, etc.

Note que deveria haver um momento em que a definição termina, pois senão seria impossível calcular n^k . Por isso, toda definição recursiva deve ser acompanhada de um caso trivial que será o final da definição. No exemplo apresentado, $n^0 = 1$ é o caso trivial e determina o final da recursividade sobre n^k.

Assim, seria possível calcular, por exemplo 3^3:

$3^3 = 3.3^2$, $3^2 = 3.3^1$, $3^1 = 3.3^0$, $3^0 = 1 \Rightarrow 3^1 = 3$ $3^2 = 3.3$ e $3^3 = 3.3.3$

Na linguagem C, funções podem chamar a si próprias, ou seja, funções podem ser recursivas também, já que podem ser definidas através delas mesmas.

Para uma linguagem permitir recursividade, uma função deve estar apta a chamar a si própria. Um exemplo clássico de recursividade em programação é a função que calcula o fatorial de um número.

–Exemplo 1: Duas versões de fatorial –

```
/* não recursiva */
int fat (int n)
{ int t, resp;
  resp = 1;
  for (t=1; t<=n; t++)
  resp = resp*t;
  return resp;
}
/* recursiva */
int fat (int n)
{ int resp;
  if (n<2) return 1;
  resp = fat(n-1)*n;
  return resp;
}
```

O funcionamento da função *fat* não recursiva deve estar claro. Ela usa uma repetição começando com 1 e terminando com o valor objetivado e multiplica progressivamente cada número pelo produto acumulado.

A operação da função *fat* recursiva é um pouco mais complexa. Quando a função *fat* é chamada com um argumento 1, a função retorna 1 (esse é o caso trivial da definição recursiva do fatorial), caso contrário, ela retorna o produto de *fat(n-1)*n*.

Para avaliar essa expressão, fat é chamada com n-1. Isso acontece até que n seja igual a 1, quando as chamadas à função começam a retornar. O exemplo abaixo ilustra a configuração da pilha na memória durante cada passo da sequência de execução da função *fat(4)*.

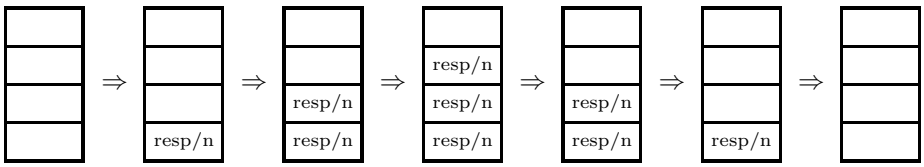


Figura E.1: Estágios da pilha na chamada recursiva de *fat(4)*

Quando uma função chama a si própria, as novas variáveis locais e os parâmetros são alocados na pilha (que é uma região da memória) e o código da função é executado com esses novos valores a partir do início. Uma chamada recursiva não faz uma nova cópia da função. Somente os argumentos e as variáveis são novas.

Quando cada chamada recursiva retorna, as antigas variáveis locais e os parâmetros são removidos da pilha e a execução recomeça no ponto de chamada da função dentro da função.

A principal vantagem das funções recursivas é que elas podem ser usadas para criar versões mais claras e mais simples de muitos algoritmos complexos do que os seus equivalentes iterativos.

Por exemplo, o algoritmo de ordenação rápida é bastante difícil de ser implementado pelo modo iterativo. Também, alguns problemas, especialmente os relacionados com IA (inteligência artificial), levam a si próprios a soluções recursivas. Finalmente, muitas definições são naturalmente recursivas, o que torna muito mais fácil implementá-las utilizando *recursividade*.

Na criação de funções recursivas é muito importante que seja definido um caso trivial que determina quando a função deverá começar a retornar valores. Se não houver um caso que obrigue a função a parar de chamar a si mesma, o programa certamente irá entrar estourar a pilha, já que a memória não é infinita.

E.1 Exercícios

1. Crie uma definição recursiva para as seguintes operações:

   a) soma de dois números a e b;

   b) multiplicação de dois números a e b ;

   c) cálculo do n-ésimo número de uma PA de razão r;

   d) cálculo do n-ésimo número de uma PG de razão q;

2. Implemente a função soma_pa (int x, int r, int n) que retorna a soma dos n termos de uma PA de termo inicial x e razão r.

3. Desenhe um diagrama da memória para a seguinte chamada de soma_pa:

```
soma_pa(1,3,4);
```

Referências Bibliográficas

[RIT86] B.W. KERNIGHAN; D. M. RITCHIE. *C, A Linguagem de Programação.* Editora Campus, Rio de Janeiro, 1986.