

Universidade Federal de Santa Catarina – UFSC
Centro Tecnológico - CTC
Departamento de Automação e Sistemas - DAS
Sistemas Industriais Inteligentes – S2i
<http://s2i.das.ufsc.br/>

Curso de Linguagem Computacional C/C++



Florianópolis, janeiro de 2002.

Índice

Índice	2
1 Programação em C	7
2 Conceitos Básicos da Programação C	9
2.1 Histórico de C.....	9
2.2 Criando um Programa Executável.....	9
2.3 A Estrutura Básica de um Programa em C.....	10
2.4 Variáveis.....	11
2.5 Tipos de Dados.....	12
2.6 Constantes	14
2.7 Ponteiros.....	15
2.8 Exercícios	16
3 Entrada/Saída Console	17
3.1 Printf()	17
3.2 Cprintf().....	19
3.3 Scanf()	19
3.4 Getch(), Getche() e Getchar().....	20
3.5 Puth() ou Puthchar()	21
3.6 Exercícios	21
4 Operadores.....	23
4.1 Operadores Aritméticos.....	23
4.2 Operadores Relacionais.....	23
4.3 Operadores lógicos binários	24
4.4 Operadores de Ponteiros.....	25
4.5 Operadores Incrementais e Decrementais	25
4.6 Operadores de Atribuição.....	27
4.7 O Operador Lógico Ternário.....	28
4.8 Precedência.....	28
4.9 Exercícios:	28
5 Laços	30
5.1 O Laço For.....	30
5.2 O Laço While	31
5.3 O Laço Do-While	32
5.4 Break e Continue	33
5.5 Goto.....	33
5.6 Exercícios	33
6 Comandos para Tomada de Decisão	34
6.1 If	34
6.2 If-Else	35
6.3 Switch.....	35
6.4 Exercícios	37
7 Funções.....	39
7.1 Sintaxe	39
7.2 Exemplos	40
7.3 Prototipagem	41
7.4 Classes de Armazenamento.....	42
7.4.1 Auto	42

7.4.2	Extern	42
7.4.3	Static.....	43
7.4.4	Variáveis Estáticas Externas	44
7.4.5	Register.....	44
7.5	Exercícios	44
8	Diretivas do Pré-Processador	46
8.1	Diretiva #define	46
8.2	Macros	47
8.3	Diretiva #undef.....	48
8.4	Diretiva #include	48
8.5	Compilação Condicional	49
8.6	Operador defined.....	49
8.7	Diretiva #error	50
8.8	diretiva #pragma	50
8.9	Exercícios	50
9	Matrizes	51
9.1	Sintaxe de Matrizes	51
9.2	Inicializando Matrizes	52
9.3	Matrizes como Argumentos de Funções	54
9.4	Chamada Por Valor e Chamada Por Referência	55
9.5	Strings.....	57
9.5.1	Strings Constantes	57
9.5.2	String Variáveis.....	57
9.5.3	Funções para Manipulação de Strings.....	58
9.6	Exercícios	60
10	Tipos Especiais de Dados.....	61
10.1	Typedef.....	61
10.2	Enumerados (Enum).....	61
10.3	Estruturas (Struct).....	62
10.4	Unões.....	65
10.5	Bitfields	66
10.6	Exercícios	67
11	Ponteiros e a Alocação Dinâmica de Memória	68
11.1	Declaração de Ponteiros e o Acesso de Dados com Ponteiros.....	68
11.2	Operações com Ponteiros	68
11.3	Funções & Ponteiros	70
11.4	Ponteiros & Matrizes.....	71
11.5	Ponteiros & Strings	73
11.6	Ponteiros para Ponteiros.....	74
11.7	Argumentos da Linha de Comando.....	77
11.8	Ponteiros para Estruturas.....	77
11.9	Alocação Dinâmica de Memória.....	78
11.9.1	Malloc()	79
11.9.2	Calloc()	81
11.9.3	Free()	81
11.10	Exercícios	81
12	Manipulação de Arquivos em C.....	83
12.1	Tipos de Arquivos	83
12.2	Declaração, abertura e fechamento	83
12.3	Leitura e escrita de caracteres	84

12.4	Fim de Arquivo (EOF)	84
12.5	Leitura e escrita de strings.....	85
12.6	Arquivos Padrões	85
12.7	Gravando um Arquivo de Forma Formatada	86
12.8	Leitura e escrita de valores binários	87
12.9	Exercícios	87
13	Programação em C++	89
13.1	Palavras-chave C++.....	91
13.2	Sintaxe & Variáveis.....	91
13.3	Laços e Comandos de Decisão	92
13.4	I/O em C++: Stream	92
13.4.1	A stream de saída cout.....	92
13.4.2	A stream de entrada cin	94
13.5	Funções.....	95
13.5.1	Valores Default Para Argumentos de uma Função	95
13.5.2	Sobrecarga de Funções	96
13.5.3	Funções Inline	96
13.5.4	Operador Unário de Referência: &.....	96
13.6	Alocação Dinâmica de Memória em C++	97
13.7	Exercícios	98
14	Classes e Objetos em C++.....	99
14.1	Tipo Classe e o Encapsulamento de Dados.....	100
14.2	Definindo Classes.....	100
14.3	Membros Privados e Públicos	101
14.4	Funções-Membro.....	102
14.5	Construtores & Destrutores	102
14.6	Criando Objetos.....	103
14.7	Atributos do Tipo: Static	104
14.8	Acessando Funções e Dados Públicos.....	105
14.9	Objetos Const	105
14.10	Tipo Objeto.....	106
14.11	Exercícios	106
15	Sobrecarga de Operadores.....	108
15.1	A Sobrecarga como uma Função Global.....	109
15.2	Limitações e Características.....	110
15.3	Sobrecarga de Operadores como Função-Membro	111
15.4	Estudo de Casos.....	112
15.5	Exercícios	119
16	Herança.....	120
16.1	Derivando uma Classe.....	121
16.2	Dados Protected.....	124
16.3	Construtores & Destrutores	124
16.4	Construtor de Cópia & Alocação Dinâmica.....	125
16.5	Chamadas a Funções	126
16.6	Herança Pública e Privada.....	126
16.7	Conversões de Tipos entre Classe-Base e Derivada	127
16.8	Níveis de Herança	127
16.9	Herança Múltipla.....	128
16.10	Exercícios	131
17	Funções Virtuais e Amigas.....	132

17.1	Funções Virtuais.....	132
17.2	Destrutores Virtuais.....	134
17.3	Classe-Base Virtual	134
17.4	Funções Amigas	135
17.5	Classes Amigas.....	137
17.6	Exercícios	137
18	Operações com Arquivos Iostream	139
18.1	Estudo de Caso	140
18.2	A função Open()	143
18.3	Testando Erros.....	143
18.4	Escrevendo e Lendo em Buffers de Caracteres.....	144
18.5	Imprimindo em Periféricos.....	145
18.6	Exercícios	145
19	Namespaces	146
19.1	Exemplo e Sintaxe.....	146
19.2	Qualified Names, Using Declarations and Directives.....	147
19.3	Prevenindo Conflitos de Nomes.....	148
19.4	Namespaces sem Nome.....	148
19.5	Apelidos para Namespace	149
19.6	Declarando Nomes	149
19.7	Atualizando Códigos Antigos	149
19.8	Namespaces são Abertos	150
20	Templates	151
20.1	ClasseTemplates.....	151
20.1.1	Especificação	151
20.1.2	Membros da Classe Template	151
20.1.3	ParâmetrosTemplate	152
20.2	Funções Templates	152
20.2.1	Function Template Arguments	153
20.2.2	Sobreescrevendo Funções Templates.....	153
20.3	Especialização	154
20.4	Derivação e Templates	154
20.5	Polimorfismo	155
21	Container	156
21.1	Iteradores	156
21.2	Tipos de Containeres.....	156
21.2.1	Containeres Sequenciais.....	156
21.2.2	Contêineres Associativos:	156
21.3	Exemplo de Container	156
22	Exceptions Handling	158
22.1	Exceção	158
22.2	Como as exceções são usadas	158
22.3	Biblioteca Except <except.h>	160
22.3.1	Terminate().....	160
22.3.2	Set_terminate()	161
22.3.3	Unexpected()	161
22.3.4	Set_unexpected()	161
	Trabalho 1.....	162
	Trabalho 2.....	162
	Trabalho 3.....	162



Trabalho 4.....	162
Referências Bibliográficas.....	166
Hot Sites	167
Anexo 1 – Metodologia para o Desenvolvimento de Softwares	168
Anexo 2 – Sumário da Modelagem UML	169

1 Programação em C

Atualmente, empregam-se cada vez mais sistemas computacionais na automatização de processos industriais. Os sistemas computacionais empregados (ver Figura 1) variam desde um simples circuito lógico digital, passando por um circuito composto por um microprocessador ou um CLP, até sistemas complexos envolvendo um ou mais microcomputadores ou até estações de trabalho. Um engenheiro que atua nesta área deve conhecer os sistemas computacionais disponíveis e ser capaz de selecionar o melhor equipamento para uma dada aplicação. Além disto, este profissional deve conseguir instalar este sistema, configurá-lo e acima de tudo programá-lo para que este execute a tarefa de automatização atendendo os requisitos industriais do sistema, como imunidade a falhas ou comportamento determinístico com restrições temporais (sistemas tempo-real). Neste contexto, a programação destes sistemas se faz de suma importância. Basicamente, a inteligência dos sistemas automatizados é implementada através de programas computacionais, comandando os componentes de hardware para executar a tarefa com o comportamento desejado.

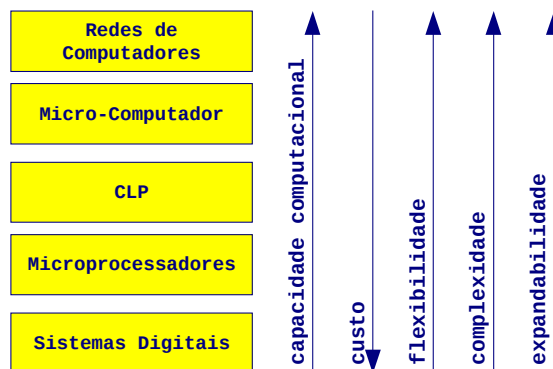


Figura 1 : Comparação entre os diversos sistemas computacionais para aplicações industriais.

Nas últimas décadas, o desenvolvimento em hardware permitiu que cada vez mais os processos industriais sejam automatizados e interligados através de sistemas computacionais. Entretanto, a evolução em software não se deu em tamanha velocidade como a de hardware. Desta forma, um dos grandes paradigmas tecnológicos hoje é o desenvolvimento de programas para a realização de tarefas complexas e que exigem um alto grau de inteligência.

A maneira de se comunicar com um computador chama-se programa e a única linguagem que o computador entende chama-se *linguagem de máquina*. Portanto todos os programas que se comunicam com a máquina devem estar em *linguagem de máquina*.

Para permitir uma maior flexibilidade e portabilidade no desenvolvimento de software, foram implementados nos anos 50 os primeiros programas para a tradução de linguagens semelhantes à humana (linguagens de "alto nível") em linguagem de máquina.

A forma como os programas são traduzidos para a linguagem de máquina classifica-se em duas categorias:

- **Interpretadores:** Um interpretador lê a primeira instrução do programa, faz uma consistência de sua sintaxe e se não houver erro converte-a para a linguagem de máquina para finalmente executá-la. Segue, então, para a próxima instrução, repetindo o processo até que a última instrução seja executada ou a consistência aponte algum erro. São muito bons para a função de depuração ("*debugging*") de programas, mas são mais lentos. Ex.: BASIC Interpretado, Java.
- **Compiladores:** Traduzem o programa inteiro em linguagem de máquina antes de serem executados. Se não houver erros, o compilador gera um programa em disco com o sufixo

.OBJ com as instruções já traduzidas. Este programa não pode ser executado até que sejam agregadas a ele rotinas em linguagem de máquina que lhe permitirão a sua execução. Este trabalho é feito por um programa chamado “*linkeditor*” que, além de juntar as rotinas necessárias ao programa **.OBJ**, cria um produto final em disco com sufixo **.EXE** que pode ser executado diretamente do sistema operacional.

Compiladores bem otimizados produzem código de máquina quase tão eficiente quanto aquele gerado por um programador que trabalhe direto em Assembly. Oferecem em geral menos facilidades de depuração que interpretadores, mas os programas são mais rápidos (na ordem de 100 vezes ou mais). Ex.: BASIC Compilado, FORTRAN, PASCAL, MÓDULA - 2, C, C++.

Além da velocidade, outras vantagens podem ser mencionadas:

- é desnecessária a presença do interpretador ou do compilador para executar o programa já compilado e linkeditado;
- programas **.EXE** não podem ser alterados, o que protege o código-fonte.

Desta forma, os compiladores requerem o uso adicional de um editor de ligações (“*Linker*”), que combina módulos-objetos (“*Traduzidos*”) separados entre si e converte os módulos assim “*linkados*” no formato carregável pelo sistema operacional (programa **.EXE**).

Estudaremos aqui uma das linguagens de alto-nível mais utilizadas na indústria: a linguagem “C” e, posteriormente, a sua sucessora a linguagem “C++”, resultado da introdução da programação “*Orientada a Objetos*” à linguagem “C”.

A programação é uma atividade que requer paciência, concentração e organização. O aprendizado desta técnica deve acontecer de forma gradual para que o programador entenda bem os conceitos envolvidos e compreenda os diversos mecanismos disponíveis. Por isso, o aluno deve prosseguir neste aprendizado no seu ritmo, permitindo-se consolidar os conhecimentos através dos exemplos e exercícios. O aluno deve buscar formar uma boa “base” antes de se preocupar com as estruturas mais complexas. A caminhada é feita passo a passo, com calma e segurança.

2 Conceitos Básicos da Programação C

2.1 Histórico de C

O compilador "C" vem se tornando o mais difundido em ambiente industrial. A linguagem "C" se originou das linguagens BCPL e B desenvolvidas em 1970. A primeira versão de "C" foi implementada para o sistema operacional UNIX pela Bell Laboratories, especificamente por Dennis M. Ritchie e Ken Thompson no início da década de 70, e rodava em um DEC PDP11 (Digital Equipment Corporation). A linguagem "C" foi utilizada para portar o UNIX para outros computadores. A linguagem "C" possui uma característica dual:

- É considerada linguagem estruturada de alto-nível e...
- "Assembly" de alto-nível, que permite escrever programas muito próximos à linguagem de máquina, sendo usada para desenvolver muitas aplicações como compiladores, interpretadores, processadores de texto e mesmo sistemas operacionais. Ex: UNIX, MS-DOS, TPW.

A linguagem de programação "C" tornou-se rapidamente uma das mais importantes e populares, principalmente por ser muito poderosa, portátil, pela padronização dos compiladores existentes (através da norma ANSI C) e flexível. Os programas em "C" tendem a ser bastante compactos e de execução rápida.

A linguagem "C" é baseada em um núcleo pequeno de funções e estruturas básicas, desta forma, todo programa é desenvolvido a partir deste núcleo básico. Isto implica na grande portabilidade de "C", haja vista que basta a implementação deste núcleo básico para um dado processador e automaticamente já estará disponível um compilador "C" para este processador. Por esta razão, existem compiladores "C" para a grande parte dos sistemas computacionais atualmente disponíveis. Devido também a este pequeno núcleo, um programador C é capaz de desenvolver programas tão eficientes, pequenos e velozes quanto os programas desenvolvidos em Assembly. Por isso, diz-se que C é uma linguagem de baixo nível, i.e., próxima da linguagem de máquina.

2.2 Criando um Programa Executável

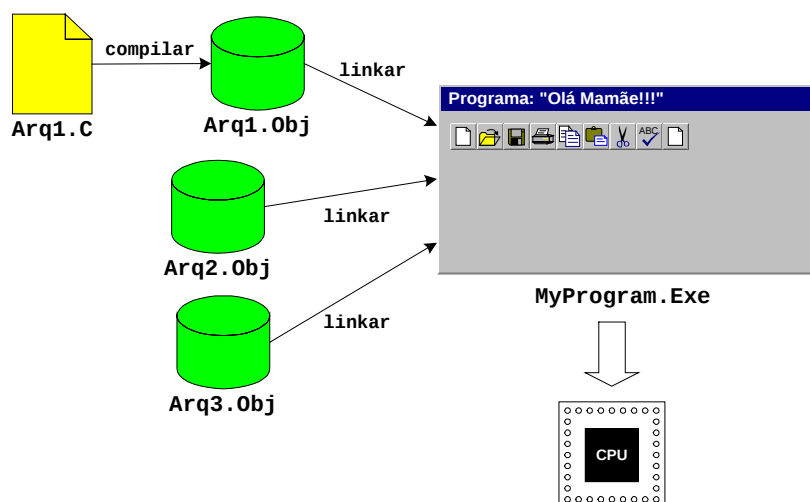


Figura 2 : Ilustração do processo de criação de um programa em C.

Primeiro, datilografe o seu programa com o auxílio de um processador de textos em modo **ASCII**. Grave o programa em disco dando a ele um nome como sufixo **.C**. O programa gerado é chamado de *código fonte*.

Na sequência, compile o fonte seguindo as instruções do seu compilador, o que criará um programa com o sufixo **.OBJ** em disco. O programa gerado é chamado de *objeto*.

Por fim, basta linkeditar o objeto seguindo as instruções do seu linkeditor o que criará um programa com sufixo **.EXE** em disco. O programa gerado é chamado de *executável*. A Figura 2 apresenta o processo de geração de um programa em C.

2.3 A Estrutura Básica de um Programa em C

A forma geral de um programa em "C" é a seguinte:

```
< diretivas do pré-processador >
< declarações globais >;
main()
{
    < declarações locais >;    /* comentário */
    < instruções >;
}
< outras funções >
```

Vamos começar por um programa bem simples em C. Você pode escrever este programa em um arquivo ASCII e salvá-lo com um nome terminando em **“.C”**. O programa serve para escrever na tela a frase **“Bom Dia!!!!”**.

```
/* Programa : Bom Dia! */
#include <stdio.h>

void main()
{
    printf("Bom Dia!!!!");
}
```

Na primeira linha, os símbolos **/*** e ***/** servem para delimitar um comentário do programa. É muito importante que os programas sejam comentados de forma organizada. Isto permite que outras pessoas possam facilmente entender o código fonte. Os comentários não são interpretados pelo compilador, servindo apenas para a documentação e esclarecimento do programador.

Depois, segue-se com uma diretiva para o pré-processador **"#include <stdio.h>"**. Isto advém do fato de C ter um núcleo pequeno de funções básicas. Ao escrever esta linha de código, o pré-processador irá acrescentar ao programa todas as funcionalidades definidas na biblioteca **"stdio"** e irá linká-la posteriormente ao programa. Desta forma, você poderá usufruir todos os serviços disponíveis nesta biblioteca.

Por fim, temos a função **"main()"**. Esta função indica ao compilador em que instrução deve ser começada a execução do programa. Portanto, esta função deve ser única, aparecendo somente uma vez em cada programa. O programa termina quando for encerrada a execução da função **main()**. No caso deste programa exemplo, ela não recebe nenhum parâmetro e também não retorna parâmetro nenhum. Isto fica explícito através da palavra-chave **void** escrita na frente do programa. Se em vez de **void** tivéssemos escrito **int**, isto significaria que a função **main()** deveria retornar um valor do tipo inteiro ao final de sua execução. Como este é o valor retornado

pela função `main()`, este também será o valor retornado pelo programa após a sua execução. As funções e as suas características serão apresentadas em detalhes nos próximos capítulos.

```
int main()
{
    printf("Bom Dia!!!!");
    return 0; /* indica ao OS que não ocorreu nenhum erro durante a
               execução do programa */
}
```

Todo o corpo de uma função em C é inicializado e finalizado através das chaves `{` e `}`. Estas chaves definem o bloco de instruções a serem executados por esta função.

A primeira instrução dada dentro do programa é `"printf("Bom Dia!!!!");"`. `Printf` é uma função definida em `"stdio.h"` para escrever dados na janela console.

Todas as instruções de programa têm que ser declaradas dentro de alguma função (na `main()` ou outra qualquer). Todas as instruções devem estar dentro das chaves que iniciam e terminam a função e são executadas na ordem em que as escrevemos.

As instruções C são sempre encerradas por um ponto-e-vírgula (`;`). O ponto-e-vírgula é parte da instrução e não um simples separador.

Esta instrução é uma chamada à função `printf()`; os parênteses nos certificam disso e o ponto-e-vírgula indica que esta é uma instrução.

Nota-se que a função é chamada escrevendo-se o nome desta e colocando-se os parâmetros desta dentro dos parênteses. A final de cada instrução em C, faz-se necessário o acréscimo de um ponto-vírgula `;"`.

As variáveis em C podem estar dentro de uma função ou no início do arquivo fonte. Variáveis declaradas no início do arquivo fonte são consideradas "globais", isto é, são visíveis (i.e. acessíveis) para todas as funções do programa. Variáveis declaradas dentro de uma função são consideradas "locais", isto é, visíveis somente pela função onde são declaradas.

"C" distingue nomes de variáveis e funções em maiúsculas de nomes em minúsculas.

Você pode colocar espaços, caracteres de tabulação e pular linhas à vontade em seu programa, pois o compilador ignora estes caracteres. Em C não há um estilo obrigatório. Entretanto, procure manter os programas tão organizados quanto for possível, pois isto melhora muito a legibilidade do programa, facilitando o seu entendimento e manutenção.

2.4 Variáveis

As variáveis são o aspecto fundamental de qualquer linguagem de computador. Uma variável em C é um espaço de memória reservado para armazenar um certo tipo de dado e tendo um nome para referenciar o seu conteúdo.

O espaço de memória de uma variável pode ser compartilhado por diferentes valores segundo certas circunstâncias. Em outras palavras, uma variável é um espaço de memória que pode conter, a cada tempo, valores diferentes.

```
/* Programa : Exemplo de variáveis! */
#include <stdio.h>

void main()
{
    int num;                /* declaracao */
    num = 2;                /* atribui um valor */
    printf("Este é o número dois: %d", num); /* acessa a variável */
}
```

```
}

```

A primeira instrução,
int num;

é um exemplo de declaração de variável, isto é, apresenta um tipo, **int**, e um nome, **num**.

A segunda instrução,
num = 2;

atribui um valor à variável e este valor será acessado através de seu nome. Usamos o operador de atribuição (=) para este fim.

A terceira instrução chama a função `printf()` mandando o nome da variável como argumento. Esta lê o valor da variável e substitui na posição indicada por **%d**, compondo assim a frase apresentada na tela. O emprego da função `printf()` será apresentado em detalhe, posteriormente.

Em C todas as variáveis devem ser declaradas.

Se você tiver mais de uma variável do mesmo tipo, poderá declará-las de uma única vez separando seus nomes por vírgulas.

```
int aviao, foguete, helicoptero;
```

2.5 Tipos de Dados

O tipo de uma variável informa a quantidade de memória, em bytes, que esta irá ocupar e a forma como o seu conteúdo será armazenado.

Em C existem apenas 5 tipos básicos de variáveis, que são:

Identificador	Categoria
char	Caracter
int	Inteiro
float	Real de ponto flutuante
double	Real de ponto flutuante de dupla precisão
void	Sem valor.

Com exceção de **void**, os tipos de dados básicos podem estar acompanhados por “*modificadores*” na declaração de variáveis. Os “*modificadores*” de tipos oferecidos em C são:

Modificadores	Efeito
signed	Bit mais significativo é usado como sinal
unsigned	Bit mais significativo é parte do número (só +)
long	Estende precisão
short	Reduz precisão

Tipos de Dados Resultantes:

Tipo	Tamanho	Valores possíveis
(signed) char	1 Byte	-128 a +127
unsigned char	1 Byte	0 a 255
(short) (signed) int	2 Bytes	-32.768 a +32.767
(short) unsigned int	2 Bytes	0 a 65.535
(signed) long int	4 Bytes	-2.147.483.648 a +2.147.483.647
unsigned long int	4 Bytes	0 a 4.294.967.295

float	4 Bytes	$\pm 3,4\text{E}-38$ a $\pm 3,4\text{E}+38$
long float	8 Bytes	$\pm 1,7\text{E}-308$ a $\pm 1,7\text{E}+308$
double	8 Bytes	$\pm 1,7\text{E}-308$ a $\pm 1,7\text{E}+308$

Observação: As declarações que aparecem na tabela acima entre parênteses (), indicam que estas declarações são optativas. Por exemplo “short unsigned int” indica a mesma precisão que “unsigned int”.

O tipo **int** tem sempre o tamanho da palavra da máquina, isto é, em computadores de 16 bits ele terá 16 bits de tamanho.

Emprega-se o **complemento de dois** dos números positivos para o cálculo e representação dos números negativos.

A escolha de nomes significativos para suas variáveis pode ajudá-lo a entender o que o programa faz e prevenir erros. Uma variável não pode ter o mesmo nome de uma palavra-chave de C. Em C, letras minúsculas e maiúsculas são diferentes.

Tabela de Palavras Chaves em C:

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

Exemplo de um programa que emprega as variáveis apresentadas.

```
void main()
{
    float y;           /* variável Real não inicializada */
    int i;             /* variável Inteira não inicializada */
    double x = 3.24;   /* variável Double inicializada com '3.24' */
    char c = 'a';      /* variável Char inicializada com 'a' */
    i = 100;           /* variável 'i' recebe o valor 100 */
    y = (float) i;     /* converte tipos */
}
```

Preste atenção na operação “y= (float) i;”. Esta operação é muito empregada para conversão de tipos de variáveis diferentes. Suponha que você tenha uma variável ‘x’ de tipo A e queira convertê-la para o tipo B e armazená-la em y deste tipo. Você pode executar esta operação através do operador:

```
y = ((B) x);
```

Atenção: Cuidado ao converter variáveis com precisão grande para variáveis com precisão pequena, ou seja, variáveis que utilizam um número diferente de bits para representar dados. Você pode perder informações importantes por causa desta conversão. Por exemplo, se você converter um float num int, você perderá todos os dígitos depois da vírgula (precisão).

```
void main()
{
    float y = 3.1415;
    int x = 0;
```

```
    x = (int) y;    /* Equivalente à: x = 3; */  
}
```

2.6 Constantes

Um constante tem valor fixo e inalterável.

No primeiro programa exemplo, mostramos o uso de uma cadeia de caracteres constante juntamente com a função `printf()`:

```
printf("Bom Dia!!!!");
```

Há duas maneiras de declarar constantes em C:

a) usando a diretiva **#define** do pré-processador:
`#define < nome da constante > < valor >`

Esta diretiva faz com que toda aparição do nome da constante no código seja substituída antes da compilação pelo valor atribuído. Não é reservado espaço de memória no momento da declaração **#define**. A diretiva deve ser colocada no início do arquivo e tem valor global (isto é, tem valor sobre todo o código).

Exemplo:

```
#define size 400  
#define true 1  
#define false 0 /* → não usa ";" nem "=" */
```

b) utilizando a palavra-chave **"const"**:

```
const < tipo > < nome > = < valor >;
```

Esta forma reserva espaço de memória para uma variável do tipo declarado. Uma constante assim declarada só pode aparecer do lado direito de qualquer equação (isto equivale a dizer que não pode ser atribuído um novo valor aquela “variável” durante a execução do programa).

Exemplo:

```
const char letra = 'a';  
const int size = 400;  
const double gravidade = 9.81;
```

```
/* Programa: Exemplo do uso de Constantes */  
#define Size 4
```

```
void main()  
{  
    const char c = 'c';  
    const int num = 10;  
    int val = Size;  
}
```

Em C uma constante caractere é escrita entre aspas simples, uma constante cadeia de caracteres entre aspas duplas e constantes numéricas com o número propriamente dito.

Exemplos de constantes:

- a) caractere: `'a'`
- b) cadeia de caracteres: `"Bom Dia !!!!!"`
- c) número: `-3.141523`

2.7 Ponteiros

Uma das mais poderosas características oferecidas pela linguagem C é o uso de ponteiros.

Um ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente. O mecanismo usado para isto é o endereço da variável. De fato, o endereço age como intermediário entre a variável e o programa que a acessa.

Basicamente, um ponteiro é uma representação simbólica de um endereço. Portanto, utiliza-se o endereço de memória de uma variável para acessá-la.

Um ponteiro tem como conteúdo um endereço de memória. Este endereço é a localização de uma outra variável de memória. Dizemos que uma variável aponta para uma outra quando a primeira contém o endereço da segunda.

A declaração de ponteiros tem um sentido diferente da de uma variável simples. A instrução:

```
int *px;
```

declara que `*px` é um dado do tipo **int** e que **px** é um ponteiro, isto é, **px** contém o endereço de uma variável do tipo **int**.

Para cada nome de variável (neste caso **px**), a declaração motiva o compilador a reservar dois bytes de memória onde os endereços serão armazenados. Além disto, o compilador deve estar ciente do tipo de variável armazenada naquele endereço; neste caso inteiro.

O endereço de uma variável pode ser passado a um ponteiro através do operador `&`, como apresentado abaixo:

```
/* Exemplo com Ponteiros */
void main()
{
    int num, valor; /* declara as variável inteira 'num' e 'valor' */
    int *ptr;       /* declara um ponteiro para um inteiro */
    ptr = 0;        /* inicializa o ponteiro com o endereço '0' */
    ptr = &num;     /* atribui ao ptr o endereço da variável num */
    num = 10;       /* atribui à variável inteira o valor '10' */
    valor = *ptr;   /* acessa o conteúdo apontado por 'ptr' e */
                  /* atribui a 'valor' */
}
```

Neste programa, primeiro declaram-se duas variáveis inteiras. Em seguida, declara-se um ponteiro para uma variável do tipo inteira.

Este ponteiro tem o seu conteúdo inicializado com '0'. Este é um procedimento normal na manipulação de ponteiros, muito empregado para evitar o aparecimento de erros. Pois, enquanto o endereço de um ponteiro for nulo, isto indicará que este endereço contém um valor inválido e, portanto, o conteúdo representado por este endereço não deve ser acessado.

Na sequência, emprega-se o operador `&` para obter o endereço da variável 'num' e armazenar este endereço em 'ptr'.

Logo após, atribuímos a variável inteira 'num' o valor 10. Como 'ptr' contém o endereço de 'num', logo 'ptr' poderá já acessar o valor 10 armazenado na variável 'num'. Isto é feito na última linha, onde o conteúdo apontado por 'ptr' é acessado e copiado para a variável 'valor'.

Outro exemplo da manipulação de ponteiros:

```
void main()
{
    int i, j, *ptr; /* declara as variáveis */
    i = 1;          /* i recebe o valor '1' */
}
```

```
j = 2;          /* j recebe o valor '2' */
ptr = &i;       /* ptr recebe o valor do endereço de i */
*ptr = *ptr + j; /* equivale a: i = i + j */
}
```

Nosso objetivo neste momento não é apresentar todas as potencialidades dos ponteiros. Estamos aqui apresentando os ponteiros primeiramente como um tipo de dado especial. O importante aqui é entender o conteúdo de um ponteiro e como este pode ser empregado. Posteriormente, apresentaremos as funcionalidades dos ponteiros à medida que formos evoluindo no aprendizado de C.

2.8 Exercícios

2.1 Crie o seu programa “Hello, World!” ou “Olá, Mamãe!”;

2.2 Crie um programa que contenha todos os tipos de variáveis possíveis e as combinações dos modificadores possíveis. Inicialize estas variáveis com valores típicos. Use o método para a conversão de tipos para converter:

- a) A variável do tipo `char` em todos os outros tipos de variáveis.
- b) A variável do tipo `double` em todos os outros tipos de variáveis.

Discuta o que acontece com a precisão das variáveis.

2.3 Crie um programa que exemplifique a utilização de ponteiros. Que contenha pelo menos a declaração de um ponteiro, sua inicialização com zero, a obtenção do endereço de uma variável com o operador ‘&’ e o acesso ao dado representado pelo ponteiro.

2.4 Utilize as diretivas `#define` para criar constantes e empregue estas constantes para inicializar as variáveis do programa acima.

2.5 Analise o seguinte programa:

```
/* Testando 1,2,3 */

#define Default_Y 2

void main()
{
    const int num = 10;
    int y = Default_Y;
    const int *ptr = 0;
    ptr = &num;
    *ptr = *ptr + y;
}
```

Aponte onde está o erro e explique por quê.

3 Entrada/Saída Console

As rotinas de entrada/saída do console se encontram nas bibliotecas "stdio.h" e "conio.h" e, por isso, estas bibliotecas devem ser incluídas nos programas aplicativos através da diretiva 'include':

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

Algumas das funções de entrada e saída para a console mais utilizadas são apresentadas a seguir:

3.1 Printf()

A função `printf()` é uma das funções de E/S (entrada e saída) que podem ser usadas em C. Ela não faz parte da definição de C mas todos os sistemas têm uma versão de `printf()` implementada. Ela permite a saída formatada na tela.

Já vimos duas aplicações diferentes da função `printf()`:

```
printf("Bom Dia!!!!");
printf("Este é o número dois: %d", num);
```

A função `printf()` pode ter um ou vários argumentos. No primeiro exemplo nós colocamos um único argumento: "Bom Dia !!!!". No segundo entretanto, colocamos dois: "Este é o número dois: %d" que está à esquerda e o valor 2 à direita da vírgula que separa os argumentos.

Sintaxe de `printf()`:

```
printf("string-formatação", < lista de parâmetros >);
```

A string de formatação pode conter caracteres que serão exibidos na tela e códigos de formatação que indicam o formato em que os argumentos devem ser impressos. No nosso segundo exemplo, o código de formatação `%d` solicita a `printf()` para imprimir o segundo argumento em formato decimal na posição da string onde aparece o `%d`.

Cada argumento deve ser separado por uma vírgula.

Além do código de formatação decimal (`%d`), `printf()` aceita vários outros. O próximo exemplo mostra o uso do código `%s` para imprimir uma cadeia de caracteres:

```
/* Exemplo da função Printf() */
#include <stdio.h>

int main(int argc, char* argv[])
{
    printf("%s esta a %d milhoes de milhas \n do sol.", "Venus", 67);
}
```

A saída será:

**Venus está a 67 milhoes de milhas
do sol.**

Aqui, além do código de formatação, a expressão de controle de `printf()` contém um conjunto de caracteres estranho: `\n`.

O `\n` é um código especial que informa a `printf()` que o restante da impressão deve ser feito em uma nova linha. A combinação de caracteres `\n` representa, na verdade, um único caractere em C, chamado de nova-linha (equivalente ao pressionamento da tecla 'Enter' em um editor de texto).

Os caracteres que não podem ser obtidos diretamente do teclado para dentro do programa (como a mudança de linha) são escritos em C, como a combinação do sinal `\` (barra invertida) com outros caracteres. Por exemplo, `\n` representa a mudança de linha.

A **string de formatação** define a forma como os parâmetros serão apresentados e tem os seguintes campos:

`"%[Flags] [largura] [.precisão] [FNlh] <tipo> [\Escape Sequence]"`

onde:

Flags	Efeito
-	justifica saída a esquerda
+	apresenta sinal (+ ou -) do valor da variável
Em branco	apresenta branco se valor positivo, sinal de - se valor negativo
#	apresenta zero no início p/ octais apresenta 0x para hexadecimais apresenta ponto decimal para reais

largura = número máximo de caracteres a mostrar

precisão = número de casas após a vírgula a mostrar

F = em ponteiros, apresentar como "Far" => base : offset (xxxx : xxxx)

N = em ponteiros, apresentar como "Near" => offset

h = apresentar como "short"

l = apresentar como "long"

Escape Sequence	Efeito
<code>\\</code>	Barra
<code>\"</code>	Aspas
<code>\0</code>	Nulo
<code>\a</code>	Tocar Sino (Bell)
<code>\b</code>	Backspace – Retrocesso
<code>\f</code>	Salta Página de Formulário
<code>\n</code>	Nova Linha
<code>\o</code>	Valor em Octal
<code>\r</code>	Retorno do Cursor
<code>\t</code>	Tabulação
<code>\x</code>	Valor em hexadecimal

Tipo	Formato
<code>%C</code>	Caracter
<code>%d, %i</code>	Inteiro decimal (signed int)
<code>%e, %E</code>	Formato científico
<code>%f</code>	Real (float)
<code>%l, %ld</code>	Decimal longo
<code>%lf</code>	Real longo (double)
<code>%o</code>	Octal (unsigned int)

%p	Pointer xxxx (offset) se Near, xxxx : xxxx (base: offset) se Far
%s	Apontador de string, emite caracteres até aparecer caracter zero (00H)
%u	Inteiro decimal sem sinal (unsigned int)
%x	Hexadecimal

Abaixo segue alguns exemplos da formatação apresentada acima:

```
/* Programa de Exemplo da formatação da saída com Printf() */  
  
#include <stdio.h>  
  
int main(int argc, char* argv[])  
{  
    float x;  
    double y = -203.4572345;  
    int a, b;  
    a = b = 12;  
    x = 3.141523;  
    printf("Bom dia");  
    printf("\n\t\tBom dia\n"); /* pula linha após escrever bom dia */  
    printf("O valor de x é %7.3f\n", x);  
    printf("Os valores de i, j e y são: %d %d %lf \n", a, b, y);  
}
```

Observação: Caso você queira imprimir na tela os caracteres especiais ‘\’ ou ‘%’, você deve escrevê-los na função `printf()` de forma duplicada, o que indicará ao compilador que este não se trata de um parâmetro da função `printf()` mas sim que deseja-se imprimir realmente este caracter. O exemplo abaixo apresenta este caso:

```
#include <stdio.h>  
void main()  
{  
    int reajuste = 10;  
    printf("O reajuste foi de %d%%. \n", reajuste);  
}  
  
a saída será:  
O reajuste foi de 10%.
```

3.2 Cprintf()

Basicamente `cprintf()` é igual a `printf()`, mas usa as coordenadas atuais do cursor e da janela que forem ajustados anteriormente, bem como ajustes de cõr de caracteres. Utilize esta função para escrever na tela em posições pré-definidas.

3.3 Scanf()

A função `scanf()` é outra das funções de E/S implementadas em todos os compiladores C. Ela é o complemento de `printf()` e nos permite ler dados formatados da entrada padrão (teclado).

A função `scanf()` suporta a entrada via teclado. Um espaço em branco ou um CR/LF (tecla “Enter”) definem o fim da leitura. Observe que isto torna inconveniente o uso de `scanf()`

para ler strings compostas de várias palavras (por exemplo, o nome completo de uma pessoa). Para realizar este tipo de tarefa, veremos outra função mais adequada na parte referente à strings.

Sua sintaxe é similar à de `printf()`, isto é, uma expressão de controle seguida por uma lista de argumentos separados por vírgula.

Sintaxe:

```
scanf("string de definição das variáveis", <endereço das variáveis>;
```

A string de definição pode conter códigos de formatação, precedidos por um sinal % ou ainda o caracter * colocado após o % que avisa à função que deve ser lido um valor do tipo indicado pela especificação, mas não deve ser atribuído a nenhuma variável (não deve ter parâmetros na lista de argumentos para estas especificações).

A lista de argumentos deve consistir nos endereços das variáveis. O endereço das variáveis pode ser obtido através do operador '&' apresentado na secção sobre ponteiros (ver 2.7).

Exemplo:

```
/* Programa exemplo do uso de Scanf() */
#include <stdio.h>

void main()
{
    float x;
    printf ("Entre com o valor de x : ");
    scanf ("%f", &x);      /* lê o valor do tipo float (%f) e armazena na
                             variável x (&x) */
}
```

Esta função funciona como o inverso da função `printf()`, ou seja, você define as variáveis que deveram ser lidas da mesma forma que definia estas com `printf()`. Desta forma, os parâmetros de `scanf()` são em geral os mesmos que os parâmetros de `printf()`, no exemplo acima observa-se esta questão.

O código de formatação de `scanf()` é igual ao código de formatação de `printf()`. Por isso, veja as tabelas de formatação de `printf()` para conhecer os parâmetros de `scanf()`.

Outro exemplo de `scanf()`:

```
/* Segundo Programa exemplo do uso de Scanf() */
#include <stdio.h>

main()
{
    float anos, dias;
    printf("Digite a sua idade em anos: ");
    scanf("%f", &anos);
    dias = anos*365;
    printf("Sua idade em dias é %.0f.\n", dias);
}
```

3.4 Getch(), Getche() e Getchar()

Em algumas situações, a função `scanf()` não se adapta perfeitamente pois você precisa pressionar [enter] depois da sua entrada para que `scanf()` termine a leitura. Neste caso, é melhor usar `getch()`, `getche()` ou `getchar()`.

A função `getchar()` aguarda a digitação de um caracter e quando o usuário apertar a tecla [enter], esta função adquire o character e retorna com o seu resultado. `getchar()` imprime na tela o character digitado.

A função `getch()` lê um único character do teclado sem ecoá-lo na tela, ou seja, sem imprimir o seu valor na tela. A função `getche()` tem a mesma operação básica, mas com eco. Ambas funções não requerem que o usuário digite [enter] para finalizar a sua execução. Veja exemplo abaixo, junto à função `putch()`

3.5 Putch() ou Putchar()

A função `putch()` apresenta um único character na tela, recebendo-o como parâmetro. A função `putchar()` executa a mesma operação, com caracteres.

Exemplo:

```
/* Exemplo das funções: getch(), getche(), putch() e putchar() */
#include <stdio.h>
#include <conio.h>

void main()
{
    char c1, c2;
    c1 = getch();          /* lê character c1 mas não mostra na tela */
    c2 = getche();         /* lê character c2, escrevendo-o na tela */
    printf("\n0 primeiro valor digitado foi: ");
    putch(c1);             /* escreve valor de c1 na tela */
    printf("\n0 segundo valor digitado foi: ");
    putchar(c2);
}
```

3.6 Exercícios

3.1 Escreva um programa que contenha uma única instrução e imprima na tela:

Esta e' a linha um. Esta e' a linha dois.
--

3.2 Escreva um programa que imprima na tela:

Um	Dois	Três.
----	------	-------

3.3 Escreva um programa que peça os dados da conta bancária de um cliente e, posteriormente, escreva na tela os dados do cliente também de forma organizada.

3.4 Escreva um programa que exemplifique todos os tipos de formatações fornecidas por `printf()`.

3.5 Escreva um programa que peça ao usuário para entrar com um valor real (float), converta este valor para uma variável do tipo char e depois imprima o seu valor na tela em formato decimal e em formato character.



3.6 Faça um programa que peça ao usuário para entrar com um caracter , converta este número para um valor inteiro e apresente na tela o valor deste número em formato float e o endereço da variável que o contém. Utilize ponteiros para armazenar e acessar este valor.

4 Operadores

4.1 Operadores Aritméticos

C é uma linguagem rica em operadores, em torno de 40. Alguns são mais usados que outros, como é o caso dos operadores aritméticos que executam operações aritméticas.

C oferece 6 operadores aritméticos binários (operam sobre dois operandos) e um operador aritmético unário (opera sobre um operando). São eles:

Binários:

=	Atribuição
+	Soma
-	Subtração
*	Multiplificação
/	Divisão
%	Módulo (devolve o resto da divisão inteira)

Unário:

-	Menos unário (indica a troca do sinal algébrico do valor)
---	---

O operador = já é conhecido dos exemplos apresentados anteriormente. Em C, o sinal de igual não tem a interpretação dada em matemática. Representa a atribuição da expressão à direita ao nome da variável à esquerda.

Os operadores + - / * representam as operações aritméticas básicas de soma, subtração, divisão e multiplicação. A seguir está um programa que usa vários operadores aritméticos e converte temperatura Fahrenheit em seus correspondentes graus Celsius.

```
/* Programa Exemplo dos operadores: + - * / */
#include <stdio.h>

void main()
{
    int ftemp = 0;
    int ctemp = 0;
    printf("Digite temperatura em graus Fahrenheit: ");
    scanf("%d", &ftemp);
    ctemp = (ftemp - 32)*5/9;
    printf("Temperatura em graus Celsius é %d", ctemp);
}
```

O operador módulo (%) aceita somente operandos inteiros. Resulta o resto da divisão do inteiro à sua esquerda pelo inteiro à sua direita. Por exemplo, 17%5 tem o valor 2 pois quando dividimos 17 por 5 teremos resto 2.

4.2 Operadores Relacionais

Os operadores relacionais são usados para fazer comparações.

Em C não existe um tipo de variável chamada “booleana”, isto é, que assuma um valor verdadeiro ou falso. O valor zero (0) é considerado falso e qualquer valor diferente de 0 é considerado verdadeiro e é representado pelo inteiro 1. Os operadores relacionais comparam dois

operandos e retornam 0 se o resultado for falso e 1 se o resultado for verdadeiro. Os operadores relacionais disponíveis em C são:

Operador C	Função
&&	E
	OU
!	NÃO
<	MENOR
<=	MENOR OU IGUAL
>	MAIOR
>=	MAIOR OU IGUAL
==	IGUAL
!=	DIFERENTE
?:	OPERADOR CONDICIONAL TERNÁRIO

O programa a seguir mostra expressões booleanas como argumento da função `printf()`:

```
/* Exemplo de Operadores Relacionais */
#include <stdio.h>

void main()
{
    int verdade, falso;
    verdade = (15 < 20);
    falso = (15 == 20);
    printf("Verdadeiro= %d, falso= %d\n", verdade, falso);
}
```

Note que o operador relacional “igual a” é representado por dois sinais de iguais. Um erro comum é o de usar um único sinal de igual como operador relacional. O compilador não o avisará que este é um erro, por quê? Na verdade, como toda expressão C tem um valor verdadeiro ou falso, este não é um erro de programa e sim um erro lógico do programador.

4.3 Operadores lógicos binários

Estes operadores são empregados para comparar os bits contidos em duas variáveis, por isso, são denominados operadores lógicos binários. Ou seja, estes operadores fazem uma comparação lógica entre cada bit dos operandos envolvidos. Os operadores binários disponíveis são:

Operador C	Função
&	E lógico
	OU lógico
^	Ou Exclusivo
~	Não
<<	Desloca esquerda
>>	Desloca direita

A seguir temos um exemplo da aplicação destes operadores.

```
/* Exemplo usando Operadores Binarios */
#include <stdio.h>
```

```
void main()
{
    int i, j, k;
    i = 1;
    j = 2;
    printf("\ti = %x (00000001)\n\tj = %x (00000010)", i, j);
    k = i & j; /* k = i AND j */
    printf("\n\t i & j => %x (00000000)", k);
    k = i ^ j; /* k = i XOR j */
    printf("\n\t i ^ j => %x (00000011)", k);
    k = i << 2; /* k = i deslocando 2 bits à esquerda (4) */
    printf("\n\t i << 2 => %x (00000100)", k);
}
```

Primeiro o programa faz uma comparação binária ‘e’(&) entre o número 1 (0x00000001) e o número 2 (0x00000010). Por isso o resultado obtido é logicamente 0, ou seja, 0x00000000.

Em seguida, o programa faz uma comparação binária ‘ou’ (^) entre estes dois números, resultando agora, logicamente, 3, ou seja, (0x00000011).

Por fim, o programa desloca o número 1 em duas casas para a esquerda, representado no programa pelo operador << 2. Assim, o resultado é o número 4 ou, em binário, (0x00000100).

A partir deste exemplo fica clara a utilização destes operadores.

4.4 Operadores de Ponteiros

Estes operadores já foram apresentados anteriormente na secção sobre ponteiros (ver 2.7). O primeiro operador serve para acessar o valor da variável, cujo endereço está armazenado em um ponteiro. O segundo operador serve para obter o endereço de uma variável.

Operador	Função
*	Acesso ao conteúdo do ponteiro
&	Obtém o endereço de uma variável

O programa abaixo apresenta novamente a aplicação destes operadores.

```
/* Operadores de Ponteiros */
void main()
{
    int i, j, *ptr;
    ptr = &i; /* atribui a ptr o endereço da variável i */
    j = *ptr; /* atribui a j o conteúdo do endereço definido por */
              /* ptr = valor de i ! */
}
```

4.5 Operadores Incrementais e Decrementais

Uma das razões para que os programas em C sejam pequenos em geral é que C tem vários operadores que podem comprimir comandos de programas. Neste aspecto, destaca-se os seguintes operadores:

Operador	Função
++	Incrementa em 1
--	Decrementa em 1

O operador de incremento (++) incrementa de um seu operando. Este operador trabalha de dois modos. O primeiro modo é chamado pré-fixado e o operador aparece antes do nome da variável. O segundo é o modo pós-fixado em que o operador aparece seguindo o nome da variável.

Em ambos os casos, a variável é incrementada. Porém quando ++**n** é usado numa instrução, **n** é incrementada antes de seu valor ser usado, e quando **n++** estiver numa instrução, **n** é incrementado depois de seu valor ser usado.

O programa abaixo mostra um exemplo destes operadores, ressaltando esta questão.

```
/* Operadores de Incremento */
#include <stdio.h>

void main()
{
    int n, m, x, y;
    n = m = 5;
    x = ++n;
    y = m++;
    printf("O resultado e:\n n = %d; x = ++n = %d;\n m = %d; y = m++ = %d;", n, x, m, y);
}
```

Vamos analisar duas expressões seguintes.

a) $k = 3 * n++$;

Aqui, primeiro 'n' é multiplicado por 3, depois o resultado é atribuído a 'k' e, finalmente, 'n' é incrementado de 1.

b) $k = 3 * ++n$;

Aqui, primeiro 'n' é incrementado em 1, depois 'n' é multiplicado por 3 e, por fim, o resultado é atribuído a 'k'.

Dica: Para evitar problemas, faça uso de parênteses para guiar a execução do programa evitando que o compilador não compreenda a ordem correta de executar a operação.

Abaixo, temos um outro exemplo dos operadores incrementais apresentado várias aplicações destes operadores:

```
/* Operadore Incrementais 2 */
void main()
{
    int i = 10;
    int *ptr = 0;
    i = i + 1; /* incrementa i */
    i++;      /* incrementa i */
    i = i - 1; /* decrementa i */
    i--;      /* decrementa i */
    ptr = &i; /* recebe o endereço de i */
    (*ptr)++; /* incrementa valor de i */
    ptr++;    /* incrementa em 2 Bytes (1 inteiro ocupa 2 Bytes) */
             /* o valor do endereço apontado pelo ponteiro ptr */
}
```

Atenção para a última instrução **ptr++**. Esta instrução irá alterar o endereço armazenado em ptr. Se o valor deste ponteiro for acessado, o dado representado por este ponteiro não tem mais nenhuma relação com a variável 'i', podendo conter qualquer dado. Por isso, tome cuidado ao

manipular ponteiros para não acessar variáveis com uso desconhecido. Isto poderá fazer com que o seu programa gere erros fatais para o sistema operacional.

4.6 Operadores de Atribuição

O operador básico de atribuição (=) pode ser combinado com outros operadores para gerar instruções em forma compacta.

Cada um destes operadores é usado com um nome de variável à sua esquerda e uma expressão à sua direita. A operação consiste em atribuir um novo valor à variável que dependerá do operador e da expressão à direita.

Se **x** é uma variável, **exp** uma expressão e **op** um operador, então:

x op= exp;

equivale a

x = (x)op(exp);

Operador C:	Função:
=	A = B
+=	A += B; ⇒ A = A + B;
-=	A -= B; ⇒ A = A - B;
*=	A *= B; ⇒ A = A * B;
/=	A /= B; ⇒ A = A / B;
%=	A %= B; ⇒ A = A % B;
>>=	A >>= B; ⇒ A = A >> B;
<<=	A <<= B; ⇒ A = A << B;
&=	A &= B; ⇒ A = A & B;
=	A = B; ⇒ A = A B;
^=	A ^= B; ⇒ A = A ^ B;

Exemplo:

```
/* Operadores de Atribuição */
#include <stdio.h>

void main()
{
    int total = 0;
    int cont = 10;
    printf("Total=%d\n", total);
    total += 1;
    printf("Total=%d\n", total);
    total ^= 2;
    printf("Total=%d\n", total);
    total <<= 2;
    printf("Total=%d\n", total);
    total *= cont;
    printf("Total=%d\n", total);
}
```

4.7 O Operador Lógico Ternário

O operador condicional possui uma construção um pouco estranha. É o único operador em C que opera sobre três expressões. Sua sintaxe geral possui a seguinte construção:

```
exp1 ? exp2 : exp3
```

A exp1 é avaliada primeiro. Se seu valor for diferente de zero (verdadeira), a exp2 é avaliada e seu resultado será o valor da expressão condicional como um todo. Se exp1 for zero, exp3 é avaliada e será o valor da expressão condicional como um todo.

Na expressão:

```
max = (a>b)?a:b
```

a variável que contém o maior valor numérico entre a e b será atribuída a max.

Outro exemplo:

```
abs = (x > 0) ? x : -x; // abs é o valor absoluto de x
```

A expressão

```
printf("%d é uma variável %s !", x, ((x%2)?"Impar":"Par");
```

imprime ímpar se x for um número “ímpar”, caso contrário imprimirá “par”.

4.8 Precedência

O operador ! é o de maior precedência, a mesma que a do operador unário. A tabela seguinte mostra as precedências dos operadores:

Operadores	Tipos
! - ++ --	Unários; não lógicos e menos aritméticos
* / %	Aritméticos
+ -	Aritméticos
< > <= >=	Relacionais
== !=	Relacionais
&&	Lógico &
	Lógico OU
= += -= *= /= %=	Atribuição

4.9 Exercícios:

4.1 Qual é o resultado da seguinte expressão:

```
int a = 1, b = 2, c = 3;  
int result = ++a/a&&!b&&c || b-- || -a+4*c>!!b;
```

4.2 Escreva uma expressão lógica que resulte 1 se o ano for bissexto e 0 se o ano não for bissexto. Um ano é bissexto se for divisível por 4, mas não por 100. Um ano também é bissexto se for divisível por 400.

4.3 Faça um programa que solicite ao usuário o ano e imprima “Ano Bissexto” ou “Ano Não-Bissexto” conforme o valor da expressão do exercício anterior. Utilize o operador condicional.

4.4 Num cercado, há vários patos e coelhos. Escreva um programa que solicite ao usuário o total de cabeças e o total de pés e determine quantos patos e quantos coelhos encontram-se nesse cercado.

4.5 Uma firma contrata um encanador a 20.000,00 por dia. Crie um programa que solicite o número de dias trabalhados pelo encanador e imprima a quantia líquida que deverá ser paga, sabendo-se que são descontados 8% para imposto de renda.

4.6 Faça um programa que solicite um caractere do teclado por meio da função `getch()`; se for uma letra minúscula imprima-a em maiúsculo, caso contrário imprima o próprio caractere. Use uma expressão condicional

4.7 Faça um programa que solicite ao usuário uma sequência binária com 16 bits. Transforme esta sequência de 0 e 1 em um número inteiro. Depois manipule este número inteiro para verificar se os bits 0, 3, 7, 14 estão habilitados (valor igual a 1). Peça a usuário se ele deseja modificar algum bit específico da palavra. Se ele quiser, modifique o bit desejado para o valor por ele fornecido.

5 Laços

Em C existem 3 estruturas principais de laços: o laço **for**, o laço **while** e o laço **do-while**.

5.1 O Laço For

O laço **for** engloba 3 expressões numa única, e é útil principalmente quando queremos repetir algo um número fixo de vezes.

Sintaxe:

```
for(incializacao ; teste ; incremento)
    instrução;
```

Os parênteses, que seguem a palavra chave for, contêm três expressões separadas por ponto-e-vírgulas, chamadas respectivamente de : “expressão de inicialização”, “expressão de teste” e “expressão de incremento”. As 3 expressões podem ser compostas por quaisquer instruções válidas em C.

- Inicialização: executada uma única vez na inicialização do laço. Serve para iniciar variáveis.
- Teste: esta é a expressão que controla o laço. Esta é testada quando o laço é iniciado ou reiniciado. Sempre que o seu resultado for verdadeiro, a instruções do laço serão executadas. Quando a expressão se tornar falsa, o laço é terminado.
- Incremento: Define a maneira como a variável de controle do laço será alterada cada vez que o laço é repetido (conta++). Esta instrução é executada, toda vez, imediatamente após a execução do corpo do laço.

```
for( conta = 0; conta < 10; conta++ )
    printf("conta=%d\n", conta);
```

Exemplo:

```
/*Laco For 1*/
#include <stdio.h>

void main()
{
    int conta;
    for(conta = 0; conta < 10; conta += 3)
        printf("conta=%d\n", conta);
}
```

Qualquer uma das expressões de um laço for pode conter várias instruções separadas por vírgulas. A vírgula é na verdade um operador C que significa “faça isto e isto”. Um par de expressões separadas por vírgula é avaliado da esquerda para a direita.

```
/* Forma Flexivel do Laço for */
#include <stdio.h>
void main()
{
    int x, y;
    for( x=0, y=0; x+y < 100; x = x+1, y++)
        printf("%d\n", x+y);
}
```

Podemos usar chamadas a funções em qualquer uma das expressões do laço.

Qualquer uma das três partes de um laço for pode ser omitida, embora os pontos-e-vírgulas devam permanecer. Se a expressão de inicialização ou a de incremento forem omitidas, elas serão simplesmente desconsideradas. Se a condição de teste não está presente é considerada permanentemente verdadeira.

O corpo do laço pode ser composto por várias instruções, desde que estas estejam delimitadas por chaves { }. O corpo do laço pode ser vazio, entretanto o ponto-e-vírgula deve permanecer para indicar uma instrução vazia.

Quando um laço está dentro de outro laço, dizemos que o laço interior está aninhado . Para mostrar esta estrutura preparamos um programa que imprime tabuada.

```
/* tabuada */
#include <stdio.h>

void main()
{
    int i,j,k;
    printf("\n");
    for(k=0; k<=1; k++)
    {
        printf("\n");
        for(i=1; i<5; i++)
            printf("Tabuada do %3d      ", i+4*k+1);
        printf("\n");
        for(i=1; i<=9; i++)
        {
            for(j = 2+4*k; j<=5+4*k; j++)
                printf("%3d x%3d = %3d      ", j, i, j*i);
            printf("\n");
        }
    }
}
```

5.2 O Laço While

O laço while utiliza os mesmos elementos que o laço for, mas eles são distribuídos de maneira diferente no programa.

Sintaxe:
while(expressão de teste)
 instrução;

Se a expressão de teste for verdadeira (diferente de zero), o corpo do laço while será executado uma vez e a expressão de teste é avaliada novamente. Este ciclo de teste e execução é repetido até que a expressão de teste se torne falsa (igual a zero), então o laço termina.

Assim como o for, o corpo do laço while pode conter uma instrução terminada por (;), nenhuma instrução desde que possua um (;) e um conjunto de instruções separadas por chaves { }.

Abaixo temos um exemplo, onde um laço while substituiu um antigo laço for.

```
/*Laco While 1*/
#include <stdio.h>

void main()
{
    int conta = 0;
```

```
while(conta < 10)
{
    printf("conta=%d\n", conta);
    conta++;
}
```

Quando se conhece a priori o número de loops que o laço deve fazer, recomenda-se o uso do laço for. Caso o número de iterações dependa das instruções executadas, então se recomenda o uso do laço while.

Os laços while podem ser aninhados como o laço for. C permite que no interior do corpo de um laço while, você possa utilizar um outro laço while.

5.3 O Laço Do-While

O laço Do-While cria um ciclo repetido até que a expressão de teste seja falsa (zero). Este laço é bastante similar ao laço while.

A diferença é que no laço do-while o teste de condição é avaliado depois do laço ser executado. Assim, o laço do-while é sempre executado pelo menos uma vez.

Sintaxe:

```
do
{
    instrução;
} while(expressão de teste);
```

Embora as chaves não sejam necessárias quando apenas uma instrução está presente no corpo do laço do-while, elas são geralmente usadas para aumentar a legibilidade.

Exemplo usando os laços while e do-while:

```
/* Programa de Exemplo do laço Do-While */
#include <stdio.h>
#include <conio.h>
#include <stdlib.h> /* requerida para rand() */

void main()
{
    char ch, c;
    int tentativas;
    do
    {
        ch = rand()%26 + 'a';
        tentativas = 1;
        printf("\nDigite uma letra de 'a' a 'Z':\n");
        while((c=getch())!= ch)
        {
            printf("%c é incorreto. Tente novamente. \n\n",c);
            tentativas++;
        }
        printf("\n%c é correto",c);
        printf("\nvoce acertou em %d tentativas", tentativas);
        printf("\nQuer jogar novamente? (s\n): ");
    }while(getche()=='s');
}
```

5.4 Break e Continue

O comando break pode ser usado no corpo de qualquer estrutura do laço C. Causa a saída imediata do laço e o controle passa para o próximo estágio do programa. Se o comando break estiver em estruturas de laços aninhados, afetará somente o laço que o contém e os laços internos a este.

O comando continue força a próxima interação do laço e pula o código que estiver abaixo. Nos while e do-while um comando continue faz com que o controle do programa vá diretamente para o teste condicional e depois continue o processo do laço. No caso do laço for, o computador primeiro executa o incremento do laço e, depois, o teste condicional, e finalmente faz com que o laço continue.

5.5 Goto

O comando goto faz com que o programa pule para a instrução logo após o label passado como parâmetro. Este comando é desnecessário e desaconselhado. Foi implementado somente para manter a compatibilidade com outros compiladores.

A instrução goto tem duas partes: a palavra goto e um nome. O nome segue as convenções de nomes de variáveis C. Por exemplo:

```
goto parte1;
```

Para que esta instrução opere, deve haver um rótulo (label) em outra parte do programa. Um rótulo é um nome seguindo por dois pontos.

```
parte1:  
    printf("Análise dos Resultados.\n");
```

A instrução goto causa o desvio do controle do programa para a instrução seguinte ao rótulo com o nome indicado. Os dois pontos são usados para separar o nome da instrução.

5.6 Exercícios

5.1 Escreva um programa que imprima os caracteres da tabela ASCII de códigos de 0 a 255. O programa deve imprimir cada caracter, seu código decima e hexadecimal. Monte uma tabela usando os parâmetros de formatação de printf().

5.2 Escreva um programa, utilizando um laço while, que leia caracteres do teclado. Enquanto o usuário não pressionar a tecla ESC de código 27, os caracteres lidos não são ecoados no vídeo. Se o caracter lido for uma letra minúscula, o programa a imprime em maiúsculo e, se for qualquer outro caracter, ele mesmo é impresso.

5.3 Elabore um programa que solicite um número inteiro ao usuário e crie um novo número inteiro com os dígitos em ordem inversa. Por exemplo o número 3567 deve ser escrito 7653.

5.4 Escreva um programa que desenhe uma moldura na tela do micro, desenhando um sol com as tuas iniciais no meio da tela e escrevendo "Feliz Natal" e o seu nome embaixo.

6 Comandos para Tomada de Decisão

Uma das tarefas fundamentais de qualquer programa é decidir o que deve ser executado a seguir. Os comandos de decisão permitem determinar qual é a ação a ser tomada com base no resultado de uma expressão condicional. Isto significa que podemos selecionar entre ações alternativas dependendo de critérios desenvolvidos no decorrer da execução do programa.

C oferece 3 principais estruturas de decisão: **if**, **if-else**, **switch**. Estas estruturas são o tema deste capítulo.

6.1 If

O comando **if** é usado para testar uma condição e caso esta condição seja verdadeira, o programa irá executar uma instrução ou um conjunto delas. Este é um dos comandos básicos para qualquer linguagem de programação.

Sintaxe:

```
if(expressão de teste)
    instrução;
```

Como C não possui variáveis booleanas, o teste sobre a condição opera como os operadores condicionais, ou seja, se o valor for igual a zero (0) a condição será falsa e se o valor for diferente de zero, a condição será verdadeira.

Como os laços, o comando **if** pode ser usado para uma única instrução ou para um conjunto delas. Caso se utilize para um conjunto de instruções, este conjunto deve ser delimitado por chaves { e }.

Um comando **if** pode estar dentro de outro comando **if**. Dizemos então que o **if** interno está aninhado.

O programa abaixo mostra exemplos do uso e da sintaxe dos comandos **if**. O primeiro **if** mostra a forma aninhada do comando **if**, o segundo apresenta a sintaxe básica e o último um exemplo onde o comando **if** executa um bloco de instruções.

```
/* Programa exemplo para o comando if */
#include <stdio.h>
#include <conio.h>
void main()
{
    char ch;
    printf("Digite uma letra de 'a' a 'Z':");
    ch = getche();
    if(ch >= 'A') /* Dois comandos if aninhados */
        if(ch <= 'Z')
            printf("\n Voce digitou um caracter valido!");

    if(ch == 'p') /* Forma basica de um comando if */
        printf("\nVoce pressionou a tecla 'p!'");

    if(ch != 'p') /* Comando if executando um bloco de instrucoes */
    {
        printf("\nVoce nao digitou a tecla 'p!'");
        printf("\n Digite qualquer caracter para finalizar.");
        ch = getche();
    }
}
```

```
}  
}
```

6.2 If-Else

No exemplo anterior o comando `if` executará uma única instrução ou um grupo de instruções, se a expressão de teste for verdadeira. Não fará nada se a expressão de teste for falsa.

O comando `else`, quando associado ao `if`, executará uma instrução ou um grupo de instruções entre chaves, se a expressão de teste do comando `if` for falsa.

O `if-else` também permite o aninhamento de outros comandos `if`, ou `if-else` dentro do bloco de instruções do `if` ou `else`.

Sintaxe:

```
if(expressão de teste)  
    instrução_1;  
else  
    instrução_2;
```

Exemplo:

```
/* Programa exemplo do If-Else */  
#include <stdio.h>  
  
void main()  
{  
    int x, y;  
  
    for(y=1; y<24; y++) /* passo de descida */  
    {  
        for(x=1; x<24; x++) /* passo de largura */  
        {  
            if(x==y || x==24-y) /* diagonal? */  
                printf("\xDB"); /* cor escura */  
            else  
                printf("\xB0"); /* cor clara */  
        }  
        printf("\n"); /* Nova Linha */  
    }  
}
```

6.3 Switch

O comando `switch` permite selecionar uma entre várias ações alternativas. Embora construções `if-else` possam executar testes para escolha de uma entre várias alternativas, muitas vezes são desleais. O comando `switch` tem um formato limpo e claro.

A instrução `switch` consiste na palavra-chave `switch` seguida do nome de uma variável ou de um número constante entre parênteses. O corpo do comando `switch` é composto de vários casos rotulados por uma constante e opcionalmente um caso `default`. A expressão entre parênteses após a palavra-chave `switch` determina para qual caso será desviado o controle do programa.

O corpo de cada caso é composto por qualquer número de instruções. Geralmente, a última instrução é `break`. O comando `break` causa a saída imediata de todo o corpo do `switch`. Na

falta do comando `break`, todas as instruções após o caso escolhido serão executadas, mesmo as que pertencem aos casos seguintes.

Sintaxe:

```
switch(variável ou constante)
{
    case constante1:
        instrução;
        instrução;
        break;
    case constante2:
        instrução;
        instrução;
        break;
    case constante3:
        instrução:
        instrução:
        break;
    default:
        instrução;
        instrução;
}
```

Você não poderá usar uma variável nem uma expressão lógica como rótulo de um caso. Pode haver nenhuma, uma ou mais instruções seguindo cada caso. Estas instruções não necessitam estar entre chaves.

O corpo de um `switch` deve estar entre chaves.

Se o rótulo de um caso for igual ao valor da expressão do `switch`, a execução começa nele. Se nenhum caso for satisfeito e existir um caso `default`, a execução começará nele. Um caso `default` é opcional.

Não pode haver casos com rótulos iguais.

A seguir apresentamos um exemplo que calcula o dia da semana a partir de uma data. O ano deve ser maior ou igual a 1600, pois nesta data houve uma redefinição do calendário.

```
/* Exemplo do comando switch */
/* Calcula o dia da semana a partir de uma data */
#include <stdio.h>
#include <conio.h>

void main()
{
    int D, M, A, i;
    long int F = 0;
    do
    {
        do
        {
            printf("\nDigite uma data na forma dd/mm/aaaa: ");
            scanf("%d/%d/%d", &D, &M, &A);
        } while(A<1600 || M<1 || M>12 || D<1 || D>31);
        F = A + D + 3*(M-1) - 1;

        if(M<3)
```

```
{
    A--;
    i = 0;
}
else
    i = .4*M+2.3;

F+= A/4 - i;
i = A/100 + 1;
i *= .75;
F -= i;
F %= 7;

switch(F)
{
    case 0:
        printf("\nDomingo");
        break;
    case 1:
        printf("\nSegunda-Feira");
        break;
    case 2:
        printf("\nTerca-Feira");
        break;
    case 3:
        printf("\nQuarta-Feira");
        break;
    case 4:
        printf("\nQuinta-Feira");
        break;
    case 5:
        printf("\nSexta-Feira");
        break;
    case 6:
        printf("\nSabado");
        break;
}
}while(getche()!=27); /* Tecla ESC nao for pressionada */
}
```

6.4 Exercícios

6.1 Crie um programa que desenhe na tela um tabuleiro de xadrez.

6.2 Crie um programa que desenhe na tela o seu quadro de horários deste semestre.

6.3 Escreva um programa que solicite ao usuário três números inteiros a, b, e c onde a é maior que 1. Seu programa deve somar todos os inteiros entre b e c que sejam divisíveis por a.

6.4 A sequência de Fibonacci é a seguinte:

1, 1, 2, 3, 5, 8, 13, 21, ...

os dois primeiros termos são iguais a 1. Cada termo seguinte é igual à soma dos dois anteriores. Escreva um programa que solicite ao usuário o número do termo e calcule o valor do



termo. Por exemplo: se o número fornecido pelo usuário for igual a 7, o programa deverá imprimir 13.

6.5 Implemente um jogo da velha, onde o programa desenha na tela um tabuleiro e você pede a dois usuários (A e B) na respectiva ordem, que jogada eles querem realizar e depois atualiza o desenho. O programa deve perceber quando o jogo acabou e anunciar o vencedor.

7 Funções

As funções servem para agrupar um conjunto de instruções de acordo com a tarefa que elas desempenham. Uma vez implementada corretamente esta tarefa, você não precisa mais se preocupar em implementar esta tarefa, basta usar a sua função. Por exemplo, quando usamos `printf()` para imprimir informações na tela, não nos preocupamos como o programa realiza esta tarefa, pois a função já fornecia este serviço de forma adequada.

As funções quando bem empregadas, facilitam bastante a organização modular do programa, permitem a reutilização de partes do programa e facilitam a sua manutenção.

Uma função é uma unidade de código de programa autônoma desenhada para cumprir uma tarefa particular. Provavelmente a principal razão da existência de funções é impedir que o programador tenha que escrever o mesmo código repetidas vezes.

7.1 Sintaxe

A estrutura de uma função C é bastante semelhante à da função `main()`. A única diferença é que `main()` possui um nome especial pois a função `main()` é a primeira a ser chamada quando o programa é executado.

Sintaxe:

```
<tipo retorno> <nome>(<tipo parâmetro> <nome parâmetro>, <...> <...>)  
{  
    <declarações locais>;  
    <comandos>;  
    return <expressão ou valor compatível com o tipo de retorno>;  
}
```

O *tipo de retorno* define o tipo de dado o qual a função retornará com o resultado de sua execução. *Nome* indica qual é o nome da função.

Em seguida temos a lista de argumentos da função inseridos entre os parênteses após o nome da função e separados por vírgulas. Os argumentos são declarados no cabeçalho da função com o *tipo parâmetro* e em seguida o *nome do parâmetro*. Uma função pode ter tantos argumentos quanto você queira definir. Quando uma função não necessita de parâmetros, o nome desta deve ser seguido por parênteses () sem nenhum parâmetro no seu interior.

A lista de instruções a serem executadas pela função vem após a lista de parâmetros e deve ser delimitada por chaves { e }. As instruções utilizam os parâmetros para executarem a tarefa.

Dentro do bloco de instruções da função, primeiramente declara-se e inicializa-se as variáveis locais e posteriormente temos os comandos da função (instruções).

O comando `return` encerra a função e retorna ao ponto do programa onde esta foi chamada. No entanto, `return` não precisa ser o último comando da função. Infelizmente, uma função em C só pode retornar um resultado e este é repassado através de `return`. Algumas linguagens como Matlab permitem que uma função retorne mais do que uma única variável contendo o resultado.

Funções com tipo de retorno `void`, não utilizam o comando `return` pois estas não retornam nenhum tipo de dado. O comando `return` é utilizado quando queremos finalizar esta função antes de executar todas as instruções. Neste caso, ele aparece sem nenhum parâmetro a sua direita, somente o ponto-e-vírgula. Quando uma função não tem um tipo de retorno definido, o compilador C considera que o tipo de retorno adotado é `void`.

Do mesmo modo que chamamos uma função de biblioteca C (`printf()`, `getche()`, ...) chamamos nossas próprias funções. Os parênteses que seguem o nome são necessários para que o compilador possa diferenciar a chamada a uma função de uma variável que você esqueceu de declarar. Visto que a chamada de uma função constitui uma instrução de programa, deve ser encerrada por ponto-e-vírgula. Entretanto, na definição de uma função, o ponto-e-vírgula não pode ser usado.

7.2 Exemplos

Abaixo apresentamos um exemplo do emprego das funções:

```
/* Beep test - Exemplo do uso de funcoes */
/* Testa a funcao doisbeep */
#include <stdio.h>

/* doisbeep() */
/* toca o auto-falante duas vezes */
doisbeep()
{
    int k;
    printf("\x7");          /* Beep 1 */
    for(k=1; k<10000; k++)   /* Gera um pequeno delay */
        ;
    printf("\x7");          /* Beep 2 */
}
/* função principal */
void main()
{
    doisbeep();
    printf("Digite um caractere: ");
    getche();
    doisbeep();
}
```

A declaração da função `doisbeep()` é equivalente a seguinte declaração:
`void doisbeep(void)`

isto acontece pois quando o tipo retornado ou o parâmetro for `void` (representando que nenhum dado será passado), estes podem ser omitidos já que C considera este tipo como default.

Em C, todos os argumentos de funções são passados “por valor”. Isto significa que à função chamada é dada uma cópia dos valores dos argumentos, e ela cria outras variáveis temporárias para armazenar estes valores. A diferença principal é que, em C, uma função chamada não pode alterar o valor de uma variável da função que chama; ela só pode alterar sua cópia temporária.

C permite a criação de funções recursivas, isto é, uma função que dentro do seu corpo de instruções `{}` existe uma chamada de função a si própria. Este caso funciona como uma chamada para uma outra função qualquer. Basta que na definição da função você faça uma chamada à própria função e você já terá implementado uma função recursiva. Entretanto, tenha cuidado com funções recursivas para não fazer com que o programa entre em loops infinitos. Exemplo:

```
long int Fatorial(long int i)
{
    if(i > 1)
        return i*Fatorial(i-1);
    else return 1;
}
```

Cada chamada recursiva da função “fatorial” coloca mais uma variável *i* do tipo **long int** na pilha (stack). É portanto necessário ter cuidado com funções recursivas, pois estas podem causar um “estouro” da pilha.

```
/* Esfera - Exemplo de funcao */
/* calcula a area da esfera */
#include <stdio.h>

float area(float r); /* declaracao do prototipo da funcao */
float potencia(float num, int pot);

void main()
{
    float raio = 0;
    printf("Digite o raio da esfera: ");
    scanf("%f", &raio);
    printf("A area da esfera e' %.2f", area(raio));
}

/* area() */
/* retorna a area da funcao */
float area(float r) /* definicao da funcao */
{
    return (4*3.14159*potencia(r,2)); /* retorna float */
}

/* potencia() */
/* eleva a uma potencia positiva um parametro dado */
float potencia(float num, int pot)
{
    float result = 0; /* declaracao de var. local */
    int i = 0;

    if(pot < 0)
        return 0; /* Indica que houve erro */
    if(pot == 0)
        return 1;

    result = num;
    for(i = 1; i < pot; i++)
        result *= num;

    return result;
}
```

7.3 Prototipagem

Toda função que for utilizada em `main()` e que for declarada após `main()`, tem que ser prototipada. O **protótipo** consiste de uma versão simplificada do cabeçalho da função, na forma:

<tipo retorno> <nome da função> (<tipo dos parâmetros>);

Se `main()` for declarado por último, nenhuma prototipagem é necessária. Protótipos são também utilizados para escrever programas compostos de vários módulos, como veremos mais à frente. Feita a declaração do protótipo da função, esta poderá ser chamada ou definida em qualquer parte do seu programa, por isso é que o protótipo de uma função tem um papel importante para a organização e flexibilidade do seu código fonte.

O programa anterior e o exemplo abaixo apresentam a forma de se criar o protótipo de uma função.

```
double Raiz(double); /* protótipo */
main()
{
    double x,y;
    x = 4.0;
    y = Raiz(x);
}

double Raiz(double valor)
{
    /* <calculo da raiz do valor>; */
    return valor;
}
```

7.4 Classes de Armazenamento

Todas as variáveis e funções em C têm dois atributos: um tipo e uma classe de armazenamento. Os tipos nós já conhecemos. As 4 classes de armazenamento serão vistas a seguir: `auto`, `extern`, `static` e `register`.

7.4.1 Auto

As variáveis que temos visto em todos os nossos exemplos estão confinadas nas funções que as usam; isto é, são visíveis ou acessíveis somente às funções onde estão declaradas. Tais variáveis são chamadas “locais” ou “automáticas”, pois são criadas quando a função é chamada e destruídas quando a função termina a sua execução.

As variáveis declaradas dentro de uma função são automáticas por “default”. Variáveis automáticas são as mais comuns dentre as 4 classes. A classe de variáveis automáticas pode ser explicitada usando-se a palavra `auto`.

As duas declarações abaixo são equivalentes:

```
auto int n;
int n;
```

7.4.2 Extern

Todas as funções C e todas as variáveis declaradas fora de qualquer função têm a classe de armazenamento `extern`. Variáveis com este atributo serão conhecidas por todas as funções declaradas depois dela. A declaração de variáveis externas é feita da mesma maneira como declaramos variáveis dentro do bloco de uma função.

Uma variável definida fora de qualquer função é dita `extern`. A palavra `extern` indica que a função usará mais uma variável externa. Este tipo de declaração, entretanto, não é obrigatório se a definição original ocorre no mesmo arquivo fonte. Abaixo temos um exemplo:

```
/* Exemplo da variavel de tipo extern */
```

```
#include <stdio.h>

/* Declaracao de variaveis extern */
int teclanum;

/* Declaracao de funcoes */
void parimpar(void);

void main()
{
    extern teclanum;

    printf("Digite teclanum: ");
    scanf("%d", &teclanum);
    parimpar();
}

/* parimpar() */
/* checa se teclanum e' par ou impar */
void parimpar(void)
{
    extern teclanum;
    if(teclanum % 2)
        printf("teclanum e' impar. \n");
    else
        printf("teclanum e' par. \n");
}
```

7.4.3 Static

Variáveis `static` de um lado se assemelham às automáticas, pois são conhecidas somente as funções que as declaram e de outro lado se assemelham às externas pois mantêm seus valores mesmo quando a função termina.

Declarações `static` têm dois usos importantes e distintos. O mais elementar é permitir a variáveis locais reterem seus valores mesmo após o término da execução do bloco onde estão declaradas.

```
/* uso do tipo static */
#include <stdio.h>

/* declaracao de prototipo */
void soma();

void main()
{
    soma();
    soma();
    soma();
}

/* soma() */
/* usa variavel static */
void soma()
{
```

```
static int i = 0;  
i++;  
printf("i = %d\n", i);  
}
```

Observe que `i` não é inicializada a cada chamada de `soma()`. Portanto, a saída será:

```
i = 0  
i = 1  
i = 2
```

7.4.4 Variáveis Estáticas Externas

O segundo e mais poderoso uso de `static` é associado a declarações externas. Junto a construções externas, permite um mecanismo de “privacidade” muito importante à programação modular.

A diferença entre variáveis externas e externas estáticas é que variáveis externas podem ser usadas por qualquer função abaixo de (a partir das) suas declarações, enquanto que variáveis externas estáticas somente podem ser usadas pelas funções de mesmo arquivo-fonte e abaixo de suas declarações. Isto é, você pode definir uma variável `static` em um arquivo fonte onde estão todas as funções que necessitam acesso a esta variável. Os demais arquivos fontes não terão acesso a ela.

7.4.5 Register

A classe de armazenamento `register` indica que a variável associada deve ser guardada fisicamente numa memória de acesso muito mais rápida chamada registrador. Um registrador da máquina é um espaço de memória junto ao microprocessador onde se podem armazenar variáveis do tipo inteiro ou char. Opera como uma variável do tipo `auto`, mas possui um tempo de acesso muito mais rápido, isto é, o programa torna-se mais rápido usando `register`. Variáveis que usualmente utilizam este tipo são as variáveis de laços e argumentos de funções, justamente, para que possamos aumentar a velocidade do programa.

Como os computadores possuem um número limitado de registradores (2 para o IBM-PC), o seu programa também só poderá utilizar um número limitado de variáveis `register`. Mas isto não nos impede de declarar quantas variáveis `register` quisermos. Se os registradores estiverem ocupados o computador simplesmente ignora a palavra `register` das nossas declarações.

```
register int i;  
for(i = 0; i < 10; i++)  
    printf("Número: %d", i);
```

7.5 Exercícios

7.1 Um número primo é qualquer inteiro positivo divisível apenas por si próprio e por 1. Escreva uma função que receba um inteiro positivo e, se este número for primo, retorne 1, caso contrário retorne 0. Faça um programa que imprima na tela os ‘n’ primeiros números primos, onde ‘n’ será fornecido pelo usuário.

7.2 Crie um programa para gerar números aleatórios, utilizando variáveis `static` para armazenar o valor da semente.

7.3 Crie um programa para calcular o fatorial de um número, para tal implemente uma função recursiva.

7.4 Escreva um programa que solicite ao usuário um ano e imprima o calendário desse ano. Utilize os programas desenvolvidos nos exercícios 4.2 e 4.3. Organize o programa de forma adequada através de funções.

7.5 Escreva uma função recursiva de nome soma() que receba um número inteiro positivo n como argumento e retorne a soma dos n primeiros números inteiros. Por exemplo, se a função recebe n = 5, deve retornar 15, pois:

$$15 = 1 + 2 + 3 + 4 + 5$$

8 Diretivas do Pré-Processador

O pré-processador C é um programa que examina o código fonte em C e executa certas modificações no código, baseado em instruções chamadas diretivas. O pré-processador faz parte do compilador e pode ser considerado uma linguagem dentro da linguagem C. Ele é executado automaticamente antes da compilação. Diretivas do pré-processador seriam instruções desta linguagem. As instruções desta linguagem são executadas antes do programa ser compilado e têm a tarefa de alterar os códigos-fonte, na sua forma de texto.

Instruções para o pré-processador devem fazer parte do texto que criamos, mas não farão parte do programa que compilamos, pois são retiradas do texto compilador antes da compilação. Diretivas do pré-processador são instruções para o compilador propriamente dito. Mais precisamente, elas operam diretamente no compilador antes do processo de compilação ser iniciado. Linhas normais de programa são instruções para o microprocessador; diretivas do pré-processador são instruções para o compilador.

Todas as diretivas do pré-processador são iniciadas com o símbolo (#). As diretivas podem ser colocadas em qualquer parte do programa, mas é costume serem colocadas no início do programa, antes de `main()`, ou antes do começo de uma função particular.

8.1 Diretiva `#define`

A diretiva `#define` pode ser usada para definir constantes simbólicas com nomes apropriados. Como por exemplo, podemos criar uma constante para conter o valor de π :

```
#define PI 3.14159
```

e depois utilizar o valor desta constante no programa:

```
area_esfera = (4*raio*raio*PI);
```

Quando o compilador encontra `#define`, ele substitui cada ocorrência de `PI` no programa por `3.14159`.

O primeiro termo após a diretiva `#define` (`PI`), que será procurada, é chamada “identificador”. O segundo termo (`3.14159`), que será substituída, é chamada “texto”. Um ou mais espaços separam o identificador do texto.

Note que só podemos escrever um comando deste por linha. Observe também que não há ponto-e-vírgula após qualquer diretiva do pré-processador.

A diretiva `#define` pode ser usada não só para definir constantes, como mostra o exemplo seguinte:

```
#include <stdio.h>
#define ERRO printf("\n Erro.\n")

void main()
{
    int i;
    printf("\nDigite um numero entre 0 a 100: ");
    scanf("%d", &i);
    if(i<0||i>100)
        ERRO;
}
```

8.2 Macros

A diretiva `#define` tem a habilidade de usar argumentos. Uma diretiva `#define` com argumentos é chamada de macro e é bastante semelhante a uma função. Eis um exemplo que ilustra como macro é definida e utilizada:

```
/* Exemplo do uso de Macros*/
#include <stdio.h>
#define PRN(n) printf("%.2f\n",n);

void main()
{
    float num1 = 27.25;
    float num2;
    num2 = 1.0/3.0;
    PRN(num1);
    PRN(num2);
}
```

Quando você usar a diretiva `#define` nunca deve haver espaço em branco no identificador.

Por exemplo, a instrução

```
#define PRN (n) printf("%.2f\n",n)
```

não trabalhará corretamente, porque o espaço entre PR e (n) é interpretado como o fim do identificador.

Para definir um texto "muito grande" devemos delimitar a linha anterior com uma barra invertida \ e prosseguir com a definição em outra linha.

Toda ocorrência de `PRN(n)` em seu programa é substituída por `printf("%.2f\n",n)`. O `n` na definição da macro é substituído pelo nome usado na chamada a macro em seu programa; portanto, `PRN(num1)` será substituído por `printf("%.2f\n", num1)`. Assim, `n` age realmente com um argumento.

Macros aumentam a clareza do código, pois permitem a utilização de nomes sugestivos para expressões.

Por segurança, coloque parênteses envolvendo o texto todo de qualquer diretiva `#define` que use argumentos, bem como envolvendo cada argumento. Por exemplo:

```
#define SOMA(x,y) x+y
ans = 10 * SOMA(3,4);
```

O compilador substituirá `SOMA(3,4)` por `3+4`, e o computador executará:

```
ans = 10 * 3+4;
```

ou seja, um erro. Para evitar isto, você deve definir a macro da seguinte forma:

```
#define SOMA(x,y) ((x)+(y))
```

Como macros são simples substituições dentro dos programas, o seu código aparecerá em cada ponto do programa onde forem usadas. Assim, a execução do programa será mais rápida que a chamada a uma função toda vez que se fizer necessário. Em contra partida, o código do programa será aumentado, pois o código da macro será duplicado cada vez que a macro for chamada. Outra vantagem do uso de macros é a não necessidade de especificar o tipo do dado.

8.3 Diretiva #undef

A diretiva `#undef` remove a mais recente definição criada com `#define`.

```
#define GRANDE 3
#define ENORME 8
#define SOMA(x,y) ((x)+(y))
...
#undef GRANDE /* cancela a definição de GRANDE */
#define ENORME 10 /* redefine ENORME para o valor 10 */
...
#undef ENORME /* ENORME volta a valer 8 */
...
#undef ENORME /* cancela a definição de ENORME */
...
#undef SOMA /* cancela a macro SOMA */
```

Observe que, para remover uma macro, somente o nome da macro deve constar na diretiva `#undef`. Não deve ser incluída a lista de argumento.

8.4 Diretiva #include

A diretiva `#include` causa a inclusão de um código fonte em outro.

Aqui está o exemplo de sua utilidade: suponha que você escreveu várias fórmulas matemáticas para calcular áreas de diversas figuras.

Os arquivos de inclusão são textos escritos em caracteres ASCII normais, criados geralmente para incorporar definições de constantes, macros, protótipos de funções, definições de tipos de dados complexos e declarações de variáveis externas. Geralmente, os arquivos de inclusão têm nome terminado com o sufixo “.H” (header ou cabeçalho).

Por exemplo, o arquivo `conio.h` contém o protótipo das funções `getch()` e `getche()` e é por esse motivo incluído nos programas que fazem uso destas funções.

Você pode colocar estas fórmulas em macros em um programa separado. No instante em que você precisar reescrevê-las para a utilização em seu programa use a diretiva `#include` para inserir este arquivo no seu código fonte. Exemplo:

```
#define PI 3.14159
#define AREA_CIRCULO(raio) (PI*(raio)*(raio))
#define AREA_RETANG(base,altura) ((base)*(altura))
#define AREA_TRIANG(base,altura) ((base)*(altura)/2)
```

Grave o programa acima como `areas.h`. Quando você quiser utilizar estas macros, basta incluir nos seus outros programas uma das diretivas:

```
#include <areas.h> /* para arquivos localizados na biblioteca do compilador */
#include "areas.h" /* para arquivos locais ou localizados na biblioteca do compilador */
```

O uso consciente das diretivas `#define` e `#include` podem melhorar e muito o desempenho e a organização dos seus projetos de software. Procure explorá-las para conhecer todo os seus potenciais.

8.5 Compilação Condicional

O pré-processador oferece diretivas que permitem a compilação condicional de um programa. Elas facilitam o desenvolvimento do programa e a escrita de códigos com maior portabilidade de uma máquina para outra ou de um ambiente a outro. São elas, as diretivas:

`#if`, `#ifdef`, `#ifndef`, `#elif`, `#else` e `#endif`

Cada diretiva `#if` deve terminar pela diretiva `#endif`. Entre `#if` e `#endif` pode ser colocado qualquer número de `#elif`, mas só se permite uma única diretiva `#else`. A diretiva `#else` é opcional e, se estiver presente, deve ser a última anterior a `#endif`.

Abaixo apresentamos um exemplo onde definimos o valor de uma constante com `#define` e depois a utilizamos para fazer uma compilação condicional :

```
#define DEBUG 1
...
#if DEBUG == 1
    printf("\nERRO = ", erro1);
#elif DEBUG == 2
    printf("\nERRO = ", erro2);
#else
    printf("\nERRO não documentado.");
#endif
```

Para testar constantes definidas com `#define` que não tenham valor nenhum, podemos utilizar `#ifdef` e `#ifndef`. Por exemplo:

```
#define VERSAO_DEMO
...
#ifdef VERSAO_DEMO
    #define NUM_REC 20
    #include "progdemo.h"
else
    #define NUM_REC MAXINT
    #include "program.h"
#endif
```

O último exemplo mostra como um único programa-fonte pode gerar dois executáveis diferentes: se a diretiva que define `VERSAO_DEMO` for inserida, o executável poderá manipular somente 20 registros e estará criada a versão demo de seu programa. Caso esta diretiva não esteja presente, o programa poderá manipular o número máximo de registros.

A diretiva `#ifndef` verifica a não-definição da constante. Por exemplo:

```
#ifndef WINDOWS
#define VERSAO "\nVersão DOS"
#else
#define VERSAO "\nVersão Windows"
#endif
```

8.6 Operador defined

Uma alternativa ao uso de `#ifdef` e `#ifndef` é o operador `defined`.

```
#if defined(UNIX) && !defined(INTEL_486)
...
#endif
```

8.7 Diretiva #error

A diretiva `#error` provoca uma mensagem de erro do compilador em tempo de compilação.

```
#if TAMANHO > TAMANHO1
#error "Tamanho incompatível"
#endif
```

8.8 diretiva #pragma

- `#pragma inline` - indica ao compilador C a presença de código Assembly no arquivo.
- `#pragma warn` - indica ao compilador C para ignorar "warnings" (avisos)

8.9 Exercícios

8.1 Escreva uma macro que tenha valor 1 se o seu argumento for um número ímpar, e o valor 0 se for par.

8.2 Escreva uma macro que encontre o maior entre seus três argumentos.

8.3 Escreva uma macro que tenha valor 1 se o seu argumento for um caractere entre '0' e '9', e 0 se não for.

8.4 Escreva uma macro que converta um dígito ASCII entre '0' e '9' a um valor numérico entre 0 e 9.

8.5 Escreva um programa que solicite ao usuário uma letra e retorne o número equivalente desta letra na tabela ASCII em decimal e hexadecimal. Utilize a compilação condicional para gerar um programa com interface para diferentes línguas, por exemplo uma para português e outra para inglês. Caso nenhuma das duas for definida, gere um erro de compilação. Dica, coloque o texto de cada língua em um header diferente. Por exemplo, os textos em português ficam definidos em *"progport.h"* e os textos em inglês em *"progingl.h"*

8.6 Faça com que o programa do exercício 7.4, escreva o calendário de um ano qualquer dado pelo usuário em três línguas diferentes: alemão, inglês e português. Utilize a mesma idéia que o programa anterior.

9 Matrizes

Uma matriz é um tipo de dado usado para representar uma certa quantidade de variáveis de valores homogêneos.

Imagine que você esteja precisando armazenar e manipular as notas de um ano inteiro de um conjunto de 40 alunos. Você certamente precisará de uma maneira conveniente para referenciar tais coleções de dados similares. Matriz é o tipo de dado oferecido por C para este propósito.

Um matriz é uma série de variáveis do mesmo tipo referenciadas por um único nome, onde cada variável é diferenciada através de um número chamado “subscrito” ou “índice”. Colchetes são usados para conter o subscrito. A declaração:

```
int notas[5];
```

aloca memória para armazenar 5 inteiros e anuncia que notas é uma matriz de 5 membros ou “elementos”.

Vamos agora fazer um programa que calcula a média das notas da prova do mês de Junho de 5 alunos.

```
/* Exemplo de matrizes          */
/* media das notas de 5 alunos  */

#include <stdio.h>
#define NUM_ALUNOS 5

void main()
{
    int notas[NUM_ALUNOS];      /* Declaracao de matrizes */
    int i, soma;
    for(i=0; i<NUM_ALUNOS; i++)
    {
        printf("Digite a nota do aluno %d: ", i);
        scanf("%d", &notas[i]); /* atribuindo valores a matriz */
    }
    soma = 0;
    for(i=0; i<NUM_ALUNOS; i++)
        soma = soma + notas[i]; /* lendo os dados da matriz */
    printf("Media das notas: %d.", soma/NUM_ALUNOS);
}
```

9.1 Sintaxe de Matrizes

As dimensões são definidas entre colchetes, após a definição convencional do tipo de variável.

```
<Tipo> <nome> [<dimensão1>] [<dimensão2>] ... ;
```

Em C, matrizes precisam ser declaradas, como quaisquer outras variáveis, para que o compilador conheça o tipo de matriz e reserve espaço de memória suficiente para armazená-la. Os elementos da matriz são guardados numa sequência contínua de memória, isto é, um seguido do outro.

O que diferencia a declaração de uma matriz da declaração de qualquer outra variável é a parte que segue o nome, isto é, os pares de colchetes [e] que envolvem um número inteiro, que

indica ao compilador o tamanho da matriz. A dimensão da matriz vai depender de quantos pares de colchetes a declaração da matriz tiver. Por exemplo:

```
int notas[5];      /* declaração de uma matriz unidimensional = vetor */
unsigned float tabela[3][2]; /* matriz bidimensional */
char cubo[3][4][6]; /* matriz tridimensional */
```

Analisando-se a primeira declaração: a palavra `int` declara que todo elemento da matriz é do tipo `int`, `notas` é o nome dado a matriz e `[5]` indica que a nossa matriz terá 5 elementos do tipo “`int`”. Por definição uma matriz é composta por elementos de um único tipo.

O acesso aos elementos da matriz é feito através do número entre colchetes seguindo o nome da matriz. Observe que este número tem um significado diferente quando referencia um elemento da matriz e na declaração da matriz, onde indica o seu tamanho.

Quando referenciamos um elemento da matriz este número especifica a posição do elemento na matriz. Os elementos da matriz são sempre numerados por índices iniciados por 0 (zero). O elemento referenciado pelo número 2

`notas[2]`

não é o segundo elemento da matriz mas sim o terceiro, pois a numeração começa em 0. Assim o último elemento da matriz possui um índice, uma unidade menor que o tamanho da matriz.

O acesso com `notas[i]` pode ser tanto usado para ler ou escrever um dado na matriz. Você pode tratar este elemento como um variável simples, no caso de `notas[i]`, como uma variável inteira qualquer.

Como proceder caso você não conheça a quantidade de termos que você precisa para a tua matriz? Aplique a idéia apresentada no programa exemplo acima, com a diretiva:

```
#define NUM_ALUNOS 5
```

para alocar um número maior de termos, de forma a possuir espaço suficiente alocado para conter todos os possíveis termos da matriz.

Atenção: A linguagem C não realiza verificação de limites em matrizes; por isto nada impede que você vá além do fim da matriz. Se você transpuser o fim da matriz durante uma operação de atribuição, então atribuirá valores a outros dados ou até mesmo a uma parte do código do programa. Isto acarretará resultados imprevisíveis e nenhuma mensagem de erro do compilador avisará o que está ocorrendo.

Como programador você tem a responsabilidade de providenciar a verificação dos limites, sempre que necessário. Assim a solução é não permitir que o usuário digite dados, para elementos da matriz, acima do limite.

9.2 Inicializando Matrizes

Em C a inicialização de uma matriz é feita em tempo de compilação. Matrizes das classes `extern` e `static` podem ser inicializadas. Matrizes da classe `auto` não podem ser inicializadas. Lembre-se de que a inicialização de uma variável é feita na instrução de sua declaração e é uma maneira de especificarmos valores iniciais pré-fixados.

O programa a seguir exemplifica a inicialização de uma matriz:

```
/* exemplo de inicialização de matrizes */
/* programa que produz troco */

#include <stdio.h>
#define LIM 5

void main()
```

```
{
    int d, valor, quant;
    static int tab[] = {50, 25, 10, 5, 1};
    printf("Digite o valor em centavos: ");
    scanf("%d", &valor);
    for(d=0; d<LIM; d++)
    {
        quant = valor/tab[d];
        printf("Moedas de %d centavos = %2d\n", tab[d], quant);
        valor %= tab[d];
    }
}
```

A instrução que inicializa a matriz é:

```
static int tab[] = {50, 25, 10, 5, 1};
```

A lista de valores é colocada entre chaves e os valores são separados por vírgulas. Os valores são atribuídos na sequência, isto é, `tab[0] = 50`, `tab[1] = 25`, `tab[2] = 10` e assim por diante. Note que não foi necessária a declaração da dimensão da matriz. Caso tivéssemos escrito explicitamente a dimensão da matriz, esta deveria ser maior ou igual a dimensão fornecida pelo colchete. Se há menos inicializadores que a dimensão especificada, os outros serão zero. É um erro ter-se mais inicializadores que o necessário.

Se em seu programa você deseja inicializar uma matriz, você deve criar uma variável que existirá durante toda a execução do programa. As classes de variáveis com esta característica são `extern` e `static`.

A linguagem C permite matrizes de qualquer tipo, incluindo matrizes de matrizes. Por exemplo, uma matriz de duas dimensões é uma matriz em que seus elementos são matrizes de uma dimensão. Na verdade, em C, o termo duas dimensões não faz sentido pois todas as matrizes são de uma dimensão. Usamos o termo mais de uma dimensão para referência a matrizes em que os elementos são matrizes.

As matrizes de duas dimensões são inicializadas da mesma maneira que as de dimensão única, isto é, os elementos (matrizes) são colocados entre as chaves depois do sinal de igual e separados por vírgulas. Cada elemento é composto por chaves e seus elementos internos separados por vírgulas. Como exemplo segue abaixo uma matriz bidimensional:

```
char tabela[3][5] =
    { {0, 0, 0, 0, 0},
      {0, 1, 1, 1, 0},
      {1, 1, 0, 0, 1} };
```

Cada elemento da matriz `tabela` na verdade será outra matriz, então por exemplo:
`tabela[2] == {1, 1, 0, 0, 1}`

Da mesma maneira se aplica para matrizes de três ou mais dimensões:

```
int tresd[3][2][4] =
    { { {1, 2, 3, 4} , {5, 6, 7, 8} },
      { {7, 9, 3, 2} , {4, 6, 8, 3} },
      { {7, 2, 6, 3} , {0, 1, 9, 4} } };
```

A matriz de fora tem três elementos, cada um destes elementos é uma matriz de duas dimensões de dois elementos onde cada um dos elementos é uma matriz de uma dimensão de quatro números.

Como podemos acessar o único elemento igual a 0 na matriz do exemplo anterior? O primeiro índice é [2], pois é o terceiro grupo de duas dimensões. O segundo índice é [1], pois é o segundo de duas matrizes de uma dimensão. O terceiro índice é [0], pois é o primeiro elemento da matriz de uma dimensão. Então, temo que a primitiva abaixo é verdade:

```
Tresd[2][1][0] == 0
```

9.3 Matrizes como Argumentos de Funções

C permite passar um matriz como parâmetro para uma função.

A seguir apresentamos como exemplo o método de Ordenação Bolha como exemplo de passagem de matrizes como argumentos. Este é um método famoso de ordenação de elementos em uma matriz devido a sua simplicidade e eficiência.

A função começa considerando a primeira variável da matriz, list[0]. O objetivo é colocar o menor item da lista nesta variável. Assim, a função percorre todos os itens restantes da lista, de list[1] a list[tam-1], comparando cada um com o primeiro item. Sempre que encontrarmos um item menor, eles são trocados. Terminada esta operação, é tomado o próximo item, list[1]. Este item deverá conter o próximo menor item. E novamente são feitas as comparações e trocas. O processo continua até que a lista toda seja ordenada.

```
/* Ordena.c */
/* ordena os valores da matriz */

#include <stdio.h>
#define TAMAX 30
void ordena(int list[], int tam);

void main()
{
    int list[TAMAX], tam=0, d;
    do
    {
        printf("Digite numero: ");
        scanf("%d", &list[tam]);
    } while(list[tam++] != 0);

    ordena(list, --tam);
    for(d=0; d<tam; d++)
        printf("%d\n", list[d]);
}

/* ordena(), ordena matriz de inteiros */
void ordena(int list[], int tam)
{
    int register j, i;
    int temp;
    for(j = 0; j < tam-1; j++)
        for(i = j + 1; i < tam; i++)
            if(list[j] > list[i])
            {
                temp = list[i];
                list[i] = list[j];
            }
}
```

```
        list[j] = temp;  
    }  
}
```

O único elemento novo no programa acima é a instrução:

```
ordena(list, --tam);
```

ela é uma chamada à função `ordena()` que ordena os elementos de uma matriz segundo o método bolha. Esta função tem dois argumentos: o primeiro é a matriz `list` e o segundo é a variável que contém o tamanho da matriz.

A matriz foi passada como argumento da função quando escrevemos somente o seu nome como parâmetro da função.

Quando desejamos referenciar um elemento da matriz devemos acessá-lo através do nome da matriz seguida de colchetes, onde o índice do elemento é escrito entre os colchetes. Entretanto se desejamos referenciar a matriz inteira, basta escrever o nome desta, sem colchetes. O nome de uma matriz desacompanhada de colchetes representa o endereço de memória onde a matriz foi armazenada.

O operador `&` não pode ser aplicado a matriz para se obter o endereço desta, pois, a endereço da matriz já está referenciado no seu nome. Portanto, a instrução abaixo seria inválida:

```
&list;
```

O operador `&` somente pode ser aplicado a um elemento da matriz. Portanto, a instrução:

```
int * ptr = &list[0];
```

será válido e irá retornar o endereço da primeira variável da matriz. Como a área de memória ocupada por uma matriz começa através do primeiro termo armazenado nesta, portanto, os endereços da matriz e do primeiro elemento serão equivalentes:

```
list == &list[0];
```

Assim, a função `ordena()` recebe um endereço de uma variável `int` e não um valor inteiro e deve declará-lo de acordo. O tipo :

```
int []
```

indica um endereço de uma variável `int`.

9.4 Chamada Por Valor e Chamada Por Referência

Quando um nome de uma simples variável é usado como argumento na chamada a uma função, a função toma o valor contido nesta variável e o instala em uma nova variável e em nova posição de memória criada pela função. Portanto as alterações que forem feitas nos parâmetros da função não terão nenhum efeito nas variações usadas na chamada. Este método é conhecido como **chamada por valor**.

Entretanto, as matrizes são consideradas um tipo de dado bastante grande, pois são formadas por diversas variáveis. Por causa disto, a linguagem C++ determina ser mais eficiente existir uma única cópia da matriz na memória, sendo portanto irrelevante o número de funções que a acessem. Assim, não são passados os valores contidos na matriz, somente o seu endereço de memória.

Desta forma, a função usa o endereço da matriz para acessar diretamente os elementos da própria matriz da função que chama. Isto significa que as alterações que forem feitas na matriz pela função afetarão diretamente a matriz original.

Quando se passa o nome de uma matriz para uma função, não se cria uma cópia dos dados desta mas sim, utiliza-se diretamente os dados da matriz original através do endereço passado. Esta tipo de passagem de parâmetros é denominado **passagem de parâmetros por referência**.

A seguir apresentamos um exemplo de passagem de uma matriz bidimensional como parâmetro de uma função. O programa exemplo avalia a eficiência de funcionários de uma loja,

quanto ao número de peças vendidas por cada um. O primeiro índice da matriz indica o número de funcionários da loja e o segundo índice, o número de meses a serem avaliados.

```
/* Exemplo da matrizes bidimensionais      */
/* histograma horizontal                    */
#include <stdio.h>

#define FUNC      4          /* numero de funcionarios */
#define MES       3          /* nuemro de meses       */
#define MAXBARRA  50         /* tamanho maximo da barra */
void grafico(int p[][MES], int nfunc); /* declaracao da funcao */

void main()
{
    int pecas[FUNC][MES]; /* declaracao de matriz bidimensional*/
    int i, j;
    for(i=0; i<FUNC; i++)
        for(j=0; j<MES; j++)
        {
            printf("Numero de pecas vendidas pelo funcionario");
            printf("%d no mes %d: ", i+1, j+1);
            scanf("%d", &pecas[i][j]);
        }
    grafico(pecas, FUNC);
}

/* grafico(), desenha um histograma */
void grafico(int p[][MES], int nfunc)
{
    int register i, j;
    int max, tam = 0, media[FUNC];
    for(i=0; i<nfunc; i++)
    {
        media[i] = 0;
        for(j=0; j<MES; j++)
            media[i] += p[i][j];
        media[i] /= MES;
    }
    max = 0;
    for(i = 0; i< nfunc; i++)
        if(media[i]>max)
            max = media[i];
    printf("\n\n\n\n\n FUNC - MEDIA\n-----\n");
    for(i=0; i<FUNC; i++)
    {
        printf("%5d - %5d : ", i+1, media[i]);
        if(media[i]>0)
        {
            if((tam=(int)((((float)media[i])*MAXBARRA/max)) <= 0)
                tam = 1;
        }
        else
            tam = 0;
        for(; tam>0; --tam)
            printf("%c", '>');
    }
}
```

```
        printf("\n");  
    }  
}
```

O método de passagem do endereço da matriz para a função é idêntico ao da passagem de uma matriz de uma dimensão, não importando quantas dimensões tem a matriz, visto que sempre passamos o seu endereço.

```
grafico(pecas, FUNC);
```

Como não há verificação de limites, uma matriz com a primeira dimensão de qualquer valor pode ser passada para a função chamada. Mas uma função que recebe uma matriz bidimensional deverá ser informada do comprimento da segunda dimensão para saber como esta matriz está organizada na memória e assim poder operar com declarações do tipo:

```
pecas[2][1]
```

pois, para encontrar a posição na memória onde `pecas[2][1]` está guardada, a função multiplica o primeiro índice (2) pelo número de elementos da segunda dimensão (MES) e adiciona o segundo índice (1), para finalmente somar ao endereço da matriz (`pecas`). Caso o comprimento da segunda matriz não seja conhecido, será impossível saber onde estão os valores.

9.5 Strings

C não possui um tipo de dado especial para tratar strings. Desta forma, uma das aplicações mais importante de matrizes em C é justamente a criação do tipo string, caracterizado por uma sequência de caracteres armazenados terminados por ‘\0’ (que possui o valor 0 na tabela ASCII) em uma matriz. Por esta razão é que C não fornece uma estrutura amigável para a manipulação de string comparada com outras linguagens como Pascal, por exemplo. Somente com o advento da orientação a objetos (C++) é que C veio a fornecer uma estrutura amigável e flexível para tratar strings. Esta estrutura será vista somente nos capítulos posteriores. Aqui, apresentaremos string na sua forma básica em C.

String é uma matriz tipo de `char` que armazena um texto formado de caracteres e sempre terminado pelo caractere zero (‘\0’). Em outras palavras, string é uma série de caracteres, onde cada um ocupa um byte de memória, armazenados em sequência e terminados por um byte de valor zero (‘\0’). Cada caracter é um elemento independente da matriz e pode ser acessado por meio de um índice.

9.5.1 Strings Constantes

Qualquer sequência de caracteres delimitadas por aspas duplas é considerada pelo compilador como uma string constante. O compilador aloca uma matriz com tamanho uma (1) unidade maior que o número de caracteres inseridos na string constante, armazenando estes caracteres nesta matriz e adicionando automaticamente o caracter delimitador de strings ‘\0’ no final da matriz. Vários exemplos de strings constantes estão presentes ao longo de texto, exemplos como :

```
printf("Bom Dia!!!");
```

O importante é lembrar que o compilador sempre adiciona o caracter ‘\0’ a strings constantes antes de armazená-las na forma de matriz para poder marcar o final da matriz. Observe que o caracter ‘\0’, também chamado de NULL, tem o valor zero decimal (tabela ASCII), e não se trata do caracter ‘0’, que tem valor 48 decimal (tabela ASCII). A terminação ‘\0’ é importante, pois é a única maneira que as funções possuem para poder reconhecer onde é o fim da string.

9.5.2 String Variáveis

Um das maneiras de receber uma string do teclado é através da função `scanf()` pelo formato %s. A função `scanf()` lê uma string até você apertar a tecla [enter], adiciona o caracter

'\0' e armazena em uma matriz de caracteres. Perceba que o endereço da matriz é passado escrevendo-se somente o nome da matriz. Abaixo apresentamos um exemplo, nele fica claro que se você digitar uma frase com um número maior de caracteres do que a matriz alocada, (`char[15]`) um erro de memória irá ocorrer. Isto se deve ao fato de `scanf()` não testar a dimensão de matriz e simplesmente adquirir uma sequência de caracteres até encontrar um caractere do [enter], espaço simples ou tabulação. Depois ela armazena a string na matriz fornecida, sem testa a dimensão da matriz. A definição de uma string segue o mesmo formato da definição de matrizes, entretanto o tipo empregado é o tipo `char`.

```
/* string 1, le uma string do teclado e imprime-a */
#include <stdio.h>
void main()
{
    char nome[15]; /* declara uma string com 15 caracteres */
    printf("Digite seu nome: ");
    scanf("%s", nome);
    printf("Saudacoes, %s.", nome);
}
```

Se você escrever Pedro Silva no programa acima, a função `scanf()` irá eliminar o segundo nome. Isto se deve por que `scanf()` utiliza qualquer espaço, quebra de linha ou tabulação para determinar o fim da leitura de uma string.

Neste caso, a função `gets()` se aplica de forma mais eficiente para a aquisição de strings. A função `gets()` adquire uma string até se pressionada a tecla [enter], adicionando o caractere '\0' para especificar o fim da string e armazená-la em uma matriz. Desta forma podemos utilizar `gets()` para ler strings com espaços e tabulações.

Para imprimir strings utilizamos as duas funções principais `printf()` e `puts()`.

O formato de impressão de strings com `printf()` nós já conhecemos, basta escrever o parâmetro `%s` no meio da string de formatação de `printf()` e posteriormente passar o nome da matriz de caracteres, terminada com '\0', como parâmetro.

`Puts()` tem uma sintaxe bem simples também. Chamando a função `puts()` e passando-se o nome da matriz de caracteres terminada por '\0' como parâmetro, teremos a string impressa na tela (saída padrão) até encontrar o caractere '\0'. Após a impressão, `puts()` executa uma quebra de linha automaticamente [enter], o que impossibilita a sua aplicação para escrever duas strings na mesma linha. Abaixo, temos um exemplo:

```
char nome[] = "Alexandre Orth";
puts(nome);
```

Em C, a forma formal da inicialização de strings ocorre da mesma maneira que uma matriz, ou seja, inicializamos termo por termo como a seguir:

```
char text[] = {'B', 'o', 'm', ' ', 'd', 'i', 'a', '!', '\0'};
```

C fornece ainda uma maneira bem simples, que foi apresentada no outro exemplo acima. Note que na forma formal de inicialização temos que adicionar explicitamente o caractere '\0' para determinar o fim da matriz. Já na forma simples, o compilador faz isto automaticamente para você.

9.5.3 Funções para Manipulação de Strings

Em C existem várias funções para manipular matrizes e tentar tornar a vida do programador um pouco mais fácil. Veremos aqui as 4 funções principais: `strlen()`, `strcat()`, `strcmp()`

e `strcpy()`. Note que estas funções exigem que o caracter ‘\0’ esteja delimitando o final da string.

A função `strlen()` aceita um endereço de string como argumento e retorna o tamanho da string armazenada a partir deste endereço até um caractere antes de ‘0’, ou seja, ela retorna o tamanho da matriz que armazena a string menos um, descontando assim o caractere de final da string ‘\0’. Sintaxe de `strlen()`:

```
strlen(string);
```

A função `strcat()` concatena duas strings, isto é, junta uma string ao final de outra. Ela toma dois endereços de strings como argumento e copia a segunda string no final da primeira e esta combinação gera uma nova primeira string. A segunda string não é alterada. Entretanto, a matriz da primeira string deve conter caracteres livres (ou seja, caracteres alocados localizados após o delimitador de final de string ‘\0’) suficientes para armazenar a segunda string. A função `strcat()` não verifica se a segunda string cabe no espaço livre da primeira. Sintaxe de `strcat()`:

```
strcat(string1, string2);
```

A função `strcmp()` compara duas strings e retorna:

```
< 0 :    no caso da string 1 < que a string 2  
0   :    no caso da string 1 = a string 2  
> 0 :    no caso da string 1 > que a string2
```

Neste contexto, “menor que” (<) ou “maior que” (>) indica que, se você colocar `string1` e `string2` em ordem alfabética, o que aparecerá primeiro será menor que o outro. Sintaxe de `strcmp()`:

```
strcmp(string1, string2);
```

A função `strcpy()` simplesmente copia uma string para outra. Note, que a `string1` deve conter espaço suficiente para armazenar os dados da `string2`. Sintaxe da função `strcpy()`:

```
strcpy(string1, string2);
```

Abaixo vamos mostrar um exemplo que apagar um caracter do interior de uma string. Neste exemplo vamos implementar a função `strdel()` que move, um espaço à esquerda, todos os caracteres que estão a direita do caracter sendo apagado.

```
/* delete, apaga caractere de uma string */  
#include <stdio.h>  
void strdel(char str[], int n);  
  
void main()  
{  
    char string[81];  
    int posicao;  
  
    printf("Digite string, [enter], posicao\n");  
    gets(string);  
    scanf("%d", &posicao);  
    strdel(string, posicao);  
    puts(string);  
}
```

```
/* strdel(), apaga um caractere de uma string */  
void strdel(char str[], int n)  
{  
    strcpy(&str[n], &str[n+1]);  
}
```

9.6 Exercícios

9.1 Crie um programa para calcular a matriz transposta de uma dada matriz. Aloque uma memória para uma matriz bidimensional com dimensão máxima de 10x10. Crie uma função para inicializar a matriz com zero. Depois questione o usuário para sob a dimensão da matriz que ele deseja calcular a transposta, considerando a dimensão máxima permitida. Depois, adquira os valores dos termos que compõem a matriz, solicitando ao usuário que forneça estes dados. Por fim, calcule a transposta da matriz e escreva na tela o resultado final da matriz.

9.2 Continue o programa anterior e agora calcule o tamanho de memória alocada para a matriz e que não está sendo utilizada. Ou seja, considerando o tipo de dado da matriz, que a matriz inicial tem tamanho 10x10 e o tamanho utilizado pelo usuário, quantos bytes de memória o computador alocou para a matriz e quantos bytes de memória o usuário realmente utilizou. Crie uma função que calcule este dado comparando duas matrizes de dimensões quaisquer. Reflita sobre este problema.

9.3 A decomposição LDU é uma decomposição famosa de matrizes aplicada para calcular inversas de matrizes e resolver problemas de equações lineares. Utilize o programa 9.1 e crie um programa que calcule a decomposição LDU de uma matriz, preferencialmente, com pivotamento.

9.4 Escreva uma função que indique quantas vezes aparece um determinado caracter em uma dada string.

9.5 Escreva uma função que localize um caracter em uma string e depois o substitua por outro.

9.6 Escreva uma função que insira um determinado character em uma determinada posição de uma string.

9.7 Escreva uma função que retire todos os caracteres brancos, tabulações ou nova linha [enter] de uma dada string.

9.8 Escreva um programa que converta todas os caracteres minúsculos de uma string para o correspondente caracter maiúsculo.

9.9 Refaça o seu exercício 6.2 para criar uma tabela com os seus horários ocupados e compromissos na semana. Armazene o valor de cada compromisso através de uma tabela de strings. Inicialize a tabela com valor 0, e solicite ao usuário que forneça o seu horário. Por fim, apresente na tela o resultado obtido.

10 Tipos Especiais de Dados

Em C podemos definir novos tipos de dados, adicionando complexidade aos tipos de dados já existentes. Com a finalidade de tornar a vida do programador mais fácil e permitir que este crie novos tipos de dados, compostos pelos tipos de dados já pré-existentis (char, int, float, double e matrizes destes tipos).

10.1 Typedef

Typedef nos apresenta a primeira forma de criar um novo tipo de dado. **Typedef** permite a definição de novos tipos de variáveis. Os novos tipos são sempre compostos de uma ou mais variáveis básicas agrupadas de alguma maneira especial.

Sintaxe:

```
typedef <tipo> <definição>;
```

Exemplo da definição de um novo tipo de dado:

```
typedef double real_array [10]; /* novo tipo */
real_array x, y;                /* declara variáveis tipo real_array */
```

No exemplo acima, ambas 'x' e 'y' foram declaradas como sendo do tipo 'real-array' e por isso, representam um vetor de valores reais (double) com capacidade para armazenar 10 dados deste tipo.

10.2 Enumerados (Enum)

Permitem atribuir valores inteiros sequenciais às constantes. Tipos enumerados são usados quando conhecemos o conjunto de valores que uma variável pode assumir. A variável deste tipo é sempre int e, para cada um dos valores do conjunto, atribuímos um nome significativo para representá-lo. A palavra enum enumera a lista de nomes automaticamente, dando-lhes números em sequência (0, 1, 2, etc.). A vantagem é que usamos estes nomes no lugar de números, o que torna o programa mais claro.

Abaixo apresentamos exemplos da aplicação de tipos enumerados e de suas definições:

```
/* Exemplo de tipos enumerados */
/* definicao de novos tipos de dados */
enum dias
{
    segunda, /* atribui: segunda = 0; */
    terca,   /* atribui: terca   = 1; */
    quarta,  /* atribui: quarta  = 2; */
    quinta,  /* atribui: quinta  = 3; */
    sexta,   /* atribui: sexta   = 4; */
    sabado,  /* atribui: sabado  = 5; */
    domingo  /* atribui: domingo = 6; */
};
enum cores
{
    verde      = 1,
    azul       = 2,
    preto, /* como não foi dado valor, recebe valor anterior + 1 = 3 */
    branco    = 7
```

```
};
enum boolean
{
    false,      /* false = 0 */
    true        /* true = 1  */
};

void main()
{
    enum dias dia1, dia2; /* declara variável "dias" do tipo enum */
    enum boolean b1;      /* declara variável do tipo enum boolean */

    dia1 = terca;
    dia2 = terca + 3;      /* dia2 = sexta */
    if (dia2 == sabado)
        b1 = true;
}
```

A palavra `enum` define um conjunto de nomes com valores permitidos para esse tipo e enumera esses nomes a partir de zero (default) ou, como em nosso programa, a partir do primeiro valor fornecido. Se fornecemos um valor a alguns nomes e não a outros, o compilador atribuirá o próximo valor inteiro aos que não tiverem valor. A sintaxe fica clara no exemplo acima.

Um variável de tipo enumerado pode assumir qualquer valor listado em sua definição. Valores não listados na sua definição não podem ser assumidos por esta variável. Tipos enumerados são tratados internamente como inteiros, portanto qualquer operação válida com inteiros é permitida com eles.

10.3 Estruturas (Struct)

Estruturas permitem definir estruturas complexas de dados, com campos de tipos diferentes. Observe que nas matrizes (arrays), todos os campos são do mesmo tipo. As structs permitem, entretanto, criar elementos semelhantes a arrays, mas composta de elementos com tipos diferentes de dados.

Por meio da palavra-chave `struct` definimos um novo tipo de dado. Definir um tipo de dado significa informar ao compilador o seu nome, o seu tamanho em bytes e o formato em que ele deve ser armazenado e recuperado na memória. Após ter sido definido, o novo tipo existe e pode ser utilizado para criar variáveis de modo similar a qualquer tipo simples.

Abaixo apresentamos as duas sintaxe que podem ser empregadas para a definição de estruturas e para a declaração de variáveis deste tipo:

Sintaxe :	Exemplo :
<pre>struct <nome> /* Definição */ { <tipo> <nome campo>; }[<variáveis deste tipo>]; /* Declaração de Variáveis */ struct <nome> <nome variável>;</pre>	<pre>struct Dados_Pessoais { char Nome[81]; int Idade; float Peso; } P1; /* Declaração de Variáveis */ struct Dados_Pessoais P2, P3;</pre>

<pre>typedef struct <nome> /* Def.*/ { <tipo> <nome campo>; } <nome tipo>; /* Declaração de variáveis/ <nome tipo> <nome variável>;</pre>	<pre>typedef struct Dados_Pessoais { char Nome [81]; int Idade; float Peso; } Pessoa; /* Declaração de variáveis/ Pessoa P1, P2;</pre>
--	--

Definir uma estrutura não cria nenhuma variável, somente informa ao compilador as características de um novo tipo de dado. Não há nenhuma reserva de memória. A palavra struct indica que um novo tipo de dado está sendo definido e a palavra seguinte será o seu nome.

Desta forma, faz-se necessário a declaração de variáveis deste tipo. Na primeira sintaxe, mostramos no exemplo como podemos declarar um variável do tipo da estrutura “Dados_Pessoais” já na definição da estrutura (P1) e como podemos criar outras variáveis deste tipo ao longo do programa (P2 e P3). Já na segunda sintaxe, a forma da declaração de variáveis deve ser sempre explícita, como mostrado no exemplo.

Uma vez criada a variável estrutura, seus membros podem ser acessados por meio do operador ponto. O operador ponto conecta o nome de uma variável estrutura a um membro desta. Abaixo, fornecemos exemplos de acesso as variáveis definidas por meio de estruturas:

```
gets(P1.Nome);  
P1.Idade = 41;  
P3.Peso = 75.6;
```

A linguagem C trata os membros de uma estrutura como quaisquer outras variáveis simples. Por exemplo, P1.Idade é o nome de uma variável do tipo int e pode ser utilizada em todo lugar onde possamos utilizar uma variável int.

A inicialização de estruturas é semelhante à inicialização de uma matriz. Veja um exemplo:

```
struct data  
{  
    int dia;  
    char mes[10];  
    int ano;  
};  
  
data natal = { 25, "Dezembro", 1994};  
data aniversario = { 30, "Julho", 1998};
```

Uma variável estrutura pode ser atribuída através do operador igual (=) a outra variável do mesmo tipo:

```
aniversario = natal;
```

Certamente constatamos que esta é uma estupenda capacidade quando pensamos a respeito: todos os valores dos membros da estrutura estão realmente sendo atribuídos de uma única vez aos correspondentes membros da outra estrutura. Isto já não é possível, por exemplo, com matrizes.

Estruturas não permitem operações entre elas, somente entre os seus membros.

Após definirmos um tipo de dado através de uma estrutura, podemos incluir este tipo de dado como membro de uma outra estrutura. Isto cria o que chamamos tipos aninhados de estruturas,

onde uma estrutura contém no seu interior como membro outra estrutura. Esta operação funciona assim como uma matriz bidimensional na verdade é uma matriz de matrizes unidimensionais.

Estruturas podem ser passadas para funções assim como qualquer outra variável, sendo declarada e utilizada da mesma forma. Funções podem também retornar uma estrutura como o resultado do seu processamento. Para tal, basta declarar o tipo de retorno de uma função como sendo um tipo declarado por uma struct. Esta é uma forma de fazer com que uma função retorne mais de um parâmetro.

Podemos criar uma matriz para estruturas assim como criamos uma matriz de um tipo qualquer, basta declarar o tipo da matriz como o tipo definido pela estrutura. A seguir apresentamos um exemplo envolvendo matrizes de estruturas, onde trataremos de uma matriz de estruturas que contém os dados necessários para representar um livro. A forma de acesso aos membros da estrutura, armazenadas na matriz fica clara no exemplo.

As funções `atoi()` e `atof()` apresentadas no exemplo abaixo servem para converter strings para um tipo inteiro ou float, respectivamente, e estão definidas em `stdlib`.

```
/* livros, mantem lista de livros na memoria */
#include <stdio.h>
#include <stdlib.h> /* para atof() e atoi() */

/* definicao de tipos */
struct list
{
    char    titulo[30];
    char    autor[30];
    int     regnum;
    double  preco;
};

/* declaracao de variaveis e funcoes*/
struct list  livro[50];
int          n = 0;
void         novonome();
void         listatotal();

void main()
{
    char ch;
    while(1)
    {
        printf("\nDigite 'e' para adicionar um livro");
        printf("\n'l' para listar todos os livros: ");
        ch = getche();
        switch(ch)
        {
            case 'e' :
                novonome();
                break;
            case 'l':
                listatotal();
                break;
            default:
                puts("\nDigite somente opcoes validas!!!!");
        }
    }
}
```

```
}

/* novonome(), adiciona um livro ao arquivo */
void novonome()
{
    char numstr[81];
    printf("\nRegistro %d. \nDigite titulo: ", n+1);
    gets(livro[n].titulo);
    printf("\nDigite autor: ");
    gets(livro[n].autor);
    printf("\nDigite o numero do livro (3 digitos): ");
    gets(numstr);
    livro[n].regnum = atoi(numstr); /* converte string p/ int */
    printf("\nDigite preco: ");
    gets(numstr);
    livro[n].preco = atof(numstr);
    n++;
}

/* listatotal(), lista os dados de todos os livros */
void listatotal()
{
    int i;
    if(!n)
        printf("\nLista vazia.\n");
    else
        for(i=0; i<n; i++)
        {
            printf("\nRegistro: %d.\n", i+1);
            printf("\nTitulo: %s.\n", livro[i].titulo);
            printf("\nAutor: %s.\n", livro[i].autor);
            printf("\nNumero do registro: %3d.\n", livro[i].regnum);
            printf("\nPreco: %4.3f. \n", livro[i].preco);
        }
}
```

10.4 Uniões

A palavra union é usada, de forma semelhante a struct, para agrupar um número de diferentes variáveis sob um único nome. Entretanto, uma union utiliza um mesmo espaço de memória a ser compartilhado com um número de diferentes membros, enquanto uma struct aloca um espaço diferente de memória para cada membro.

Em outras palavras, uma union é o meio pelo qual um pedaço de memória ora é tratado como uma variável de um certo tipo, ora como outra variável de outro tipo.

Por isso, uniões são usadas para poupar memória e o seu uso não é recomendado a não ser em casos extremamente necessários, pois a utilização irresponsável das uniões podem causar a perda de dados importantes do programa. Quando você declara uma variável do tipo union, automaticamente é alocado espaço de memória suficiente para conter o seu maior membro.

Sintaxe:

```
union <nome>
{
    <tipo> <campo1>;
    :
```

```
    <tipo n> <campo n>;  
};
```

Exemplo do emprego das uniões:

```
typedef unsigned char byte;  
union Oito_bytes  
{  
    double x;          /*um double de 8 bytes */  
    int    i[4];        /* um array com 4 inteiros = 8 bytes */  
    byte   j[8];        /* um array de 8 bytes */  
} unionvar;  
  
void main()  
{  
    unionvar.x = 2.7;  
    unionvar.i[1] = 3; /* sobreescreve valor anterior ! */  
}
```

Para verificar que o tamanho de uma variável union é igual ao tamanho do maior membro, vamos utilizar a função `sizeof()`. A função `sizeof()` resulta o tamanho em bytes ocupado por uma dada variável. Veja o teste realizado abaixo, onde apresentamos o valor em bytes ocupado por cada membro da união e o valor em bytes ocupado pela união inteira. Podemos verificar também que todos os membros da união ocupam a mesma posição da memória utilizando o operador `&` para obter a posição da memória onde estão armazenados os dados.

```
/* Sizeof(), mostra o uso de sizeof() */  
#include <stdio.h>  
  
union num  
{  
    char str[20];  
    int i;  
    float f;  
} x; // Cria a variavel do tipo union num.  
  
void main()  
{  
    printf("Sizeof(str[20]) = %d bytes, mem: %p.\n", sizeof(x.str), x.str);  
    printf("Sizeof(i) = %d bytes, \tmem: %p.\n", sizeof(x.i), &(x.i));  
    printf("Sizeof(f) = %d bytes, \tmem: %p.\n", sizeof(x.f), &(x.f));  
    printf("Sizeof(x) = %d bytes, \tmem: %p.\n", sizeof(x), &(x));  
}
```

A saída do programa será como esperado:

```
Sizeof(str[20]) = 20 bytes, mem: 50EF:0D66.  
Sizeof(i) = 2 bytes,          mem: 50EF:0D66.  
Sizeof(f) = 4 bytes,          mem: 50EF:0D66.  
Sizeof(x) = 20 bytes,          mem: 50EF:0D66.
```

10.5 Bitfields

Bitfields são um tipo especial de estrutura cujos campos tem comprimento especificado em bits. Estas são úteis quando desejamos representar dados que ocupam somente um bit.

Sintaxe:

```
struct <nome>
{
    <tipo> <campo> : <comprimento em bits>;
};
```

Exemplo aplicando bitfields:

```
#define ON 1
#define OFF 0

struct Registro_Flags    /* declara bitfield */
{
    unsigend int Bit_1 : 1;
    unsigend int Bit_2 : 1;
    unsigend int Bit_3 : 1;
    unsigend int Bit_4 : 1;
    unsigend int Bit_5 : 1;
    unsigend int Bit_6 : 1;
    unsigend int Bit_7 : 1;
    unsigend int Bit_8 : 1;
};
main( )
{
    struct Registro_Flags  Flags;    /* declara variável Flags */
    Flags.Bit_1 = ON;
    Flags.Bit_2 = OFF;
}
```

10.6 Exercícios

10.1 Escreva uma estrutura para descrever um mês do ano. A estrutura deve ser capaz de armazenar o nome do mês, a abreviação em letras, o número de dias e o número do mês. Escreva também um tipo enumerado para associar um nome ou uma abreviação ao número do mês.

10.2 Declare uma matriz de 12 estruturas descritas na questão anterior e inicialize-a com os dados de um ano não-bissexto.

10.3 Escreva uma função que recebe o número do mês como argumento e retorna o total de dias do ano até aquele mês. Escreva um programa que solicite ao usuário o dia, o mês e o ano. Imprima o total de dias do ano até o dia digitado.

10.4 Escreva um programa para cadastrar livros em uma biblioteca e fazer consultas a estes.

10.5 Refaça o programa do exercício 8.6 utilizando as estruturas aqui desenvolvidas.

10.6 Crie uma estrutura para descrever restaurantes. Os membros deve armazenar o nome, o endereço, o preço médio e o tipo de comida. Crie uma matriz de estruturas e escreva um programa que utilize uma função para solicitar os dados de um elemento da matriz e outra para listar todos os dados.

10.7 Crie um programa que faça o controle de cadastro de usuários e controle de alocação para uma locadora de fitas de vídeo. Utilize uma estrutura para os clientes e outra para as fitas.


```
        printf("py - px = %u\n", py-px);
    else
        printf("px - py = %u\n", px-py);

    printf("px = %p, \t*px = %d, \t&px = %p\n", px, *px, &px);
    printf("py = %p, \t*py = %d, \t&py = %p\n\n", py, *py, &py);
    px++;
    printf("px = %p, \t*px = %d, \t&px = %p\n\n", px, *px, &px);
    py = px + 3;
    printf("py = %p, \t*py = %d, \t&py = %p\n", py, *py, &py);
    printf("py - px = %u\n", py-px);
}
```

Resultado do programa:

Endereço contido inicialmente em:

px = 21CD:FFFF

py = 0000:0000

Cuidado ao utilizar ponteiros não inicializados como px!!!!

px - py = 1

px = 4F97:2126, *px = 5, &px = 4F97:2120

py = 4F97:2124, *py = 6, &py = 4F97:211C

px = 4F97:2128, *px = 20375, &px = 4F97:2120

py = 4F97:212E, *py = 20351, &py = 4F97:211C

py - px = 3

Primeiramente, o programa exemplo acima declara as variáveis inteiras ‘x’ e ‘y’, declarando dois ponteiros para inteiros também. Note que um ponteiro foi inicializado com ‘0’ e o outro não. Para mostrar a importância da inicialização de ponteiros, resolvemos mostrar na tela o valor inicial destes. Caso, tentássemos acessar o valor da variável apontada pelo ponteiro ‘py’, o programa não executaria esta operação pelo fato do ponteiro conter um endereço inválido ‘0’. Entretanto, se tentarmos acessar o valor da variável apontada por ‘px’, o computador executará esta operação sem problema, pois o endereço está válido, e assim poderemos cometer um erro gravíssimo acessando áreas de memórias inadequadas ou proibidas.

Inicialize sempre os seus ponteiros com o valor zero ou ‘NULL’.

Em seguida, atribuímos um valor válido aos ponteiros utilizando o operador (&) para obter o endereço das variáveis ‘x’ e ‘y’.

O operador (*) aplicado na frente do nome do ponteiro, retornará o valor da variável apontada por este ponteiro.

Como todas as variáveis, os ponteiros variáveis têm um endereço e um valor. O operador (&) retorna a posição de memória onde o ponteiro está localizado. Em resumo:

- O nome do ponteiro retorna o endereço para o qual ele aponta.
- O operador & junto ao nome do ponteiro retorna o endereço onde o ponteiro está armazenado.
- O operador * junto ao nome do ponteiro retorna o conteúdo da variável apontada.

No programa acima, apresentamos esta idéia de forma clara, mostrando na tela estes valores para os dois respectivos ponteiros.

Podemos incrementar um ponteiro através de adição regular ou pelo operador de incremento. Incrementar um ponteiro acarreta a movimentação do mesmo para o próximo tipo

apontado. Por exemplo, se `px` é um ponteiro para um inteiro e `px` contém o endereço `0x3006`. Após a execução da instrução:

```
px++;
```

o ponteiro `px` conterá o valor `0x3008` e não `0x3007`, pois a variável inteira ocupa 2 bytes. Cada vez que incrementamos `px` ele apontará para o próximo tipo apontado, ou seja, o próximo inteiro. O mesmo é verdadeiro para decremento. Se `px` tem valor `0x3006` depois da instrução:

```
px--;
```

ele terá valor `0x3004`.

Você pode adicionar ou subtrair de e para ponteiros. A instrução:

```
py = px + 3;
```

fará com que `py` aponte para o terceiro elemento do tipo apontado após `px`. Se `px` tem valor `0x3000`, depois de executada a instrução acima, `py` terá o valor `0x3006`.

Novamente, tenha cuidado na manipulação de ponteiros para não acessar regiões inválidas de memória ou ocupadas por outros programas.

Você pode encontrar a diferença entre dois ponteiros. Esta diferença será expressa na unidade tipo apontado, então, se `py` tem valor `0x4008` e `px` o valor `0x4006`, a expressão

```
py - px
```

terá valor 1 quando `px` e `py` são ponteiros para `int`.

Testes relacionais com `>=`, `<=`, `>` e `<` são aceitos entre ponteiros somente quando os dois operandos são ponteiros. Outro cuidado a se tomar é quanto ao tipo apontado pelo operandos. Se você comparar ponteiros que apontam para variáveis de tipo diferentes, você obterá resultados sem sentido.

Variáveis ponteiros podem ser testadas quanto à igualdade (`==`) ou desigualdade (`!=`) onde os dois operandos são ponteiros (`px == py`) ou um dos operandos `NULL` (`px != NULL` ou `px != 0`).

11.3 Funções & Ponteiros

Um das maneiras mais importantes para passar argumentos para uma função ou para que a função retorne um resultado é através de ponteiros. Ponteiros são empregados em funções principalmente quando necessitamos de umas das seguintes características:

- ponteiros permitem que uma função tenha acesso direto as variáveis da função chamadora, podendo efetuar modificações nestas variáveis;
- para tipos de dados complexos ou de grande dimensão (estruturas, matrizes, ...) é mais fácil e mais rápido acessar estes dados indiretamente através de um ponteiro do que realizar uma cópia destes para que a função possa utilizar o valor copiado.
- Podemos utilizar ponteiros de estruturas complexas ou matrizes como forma de fazer com que funções retornem mais de um valor como resultado;
- A passagem de parâmetro com ponteiros geralmente é mais rápida e eficiente do que com o próprio dado, haja visto que o compilador não faz uma cópia deste dado e um ponteiro necessita somente de 4 bytes para armazenar o seu valor.

Para tal, basta que em vez da função chamadora passar valores para a função chamada, esta passe endereços usando operadores de endereços (`&`). Estes endereços são de variáveis da função chamadora onde queremos que a função coloque os novos valores. Estes endereços devem ser armazenados em variáveis temporárias da função chamada para permitir posteriormente o seu acesso.

No exemplo a seguir mostramos como ponteiros podem ser usados como parâmetros da função.

```
/* testa funcao que altera dois valores */
#include <stdio.h>
void altera(int *, int *);

void main()
{
    int x = 0, y = 0;
    altera(&x, &y);
    printf("O primeiro e %d, o segundo e %d.", x, y);
}

/* altera(), altera dois numeros da funcao que chama*/
void altera(int *px, int *py)
{
    if(px != 0)
        *px = 3;
    if(py != 0)
        *py = 5;
}
```

Note, que o objetivo da função é prover modificações nas duas variáveis x e y. Numa forma simples, só poderíamos efetuar esta operação utilizando-se uma estrutura para conter x e y, ou criando uma função para alterar cada parâmetro. Entretanto, passando-se o endereço das variáveis x e y permitimos a função chamada que altera-se diretamente os seus valores.

O valor do endereço das variáveis foi obtido com o operador &. A função necessitou apenas da declaração de um tipo ponteiro em seu cabeçalho para estar apta a receber endereços de variáveis como um de seus parâmetros.

Uma vez conhecidos os endereços e os tipos das variáveis do programa chamador, a função pode não somente colocar valores nestas variáveis como também tomar o valor já armazenado nelas. Ponteiros podem ser usados não somente para que a função passe valores para o programa chamador, mas também para que o programa chamador passe valores para a função.

Outro aspecto importante é o teste efetuado antes de utilizar a o ponteiro recebido como parâmetro:

```
if(px != 0)
    *px = 3;
```

Isto serve para que a função verifique se o valor de endereço recebido é válido. Tome muito cuidado com a manipulação de ponteiros, um erro no programa ou em uma função do sistema operacional, pode gerar um ponteiro inválido e o acesso a área de memória representada por este ponteiro pode causar um erro fatal no seu programa. Este é um dos métodos para prever erros com ponteiros.

11.4 Ponteiros & Matrizes

Você não percebe, mas C trata matrizes como se fossem ponteiros. O tipo matriz é na verdade uma forma mais amigável que C fornece para tratar um ponteiro que aponta para uma lista de variáveis do mesmo tipo.

O compilador transforma matrizes em ponteiros quando compila, pois a arquitetura do microcomputador entende ponteiros e não matrizes.

Qualquer operação que possa ser feita com índices de uma matriz pode ser feita com ponteiros. O nome de uma matriz é um endereço, ou seja, um ponteiro. Ponteiros e matrizes são idênticos na maneira de acessar a memória. **Na verdade, o nome de uma matriz é um ponteiro constante. Um ponteiro variável é um endereço onde é armazenado um outro endereço.**

Suponha que declaramos uma matriz qualquer :

```
int table = {10, 1, 8, 5, 6}
```

e queremos acessar o terceiro elemento da matriz. Na forma convencional por matriz, fazemos isto através da instrução:

```
table[2]
```

mas a mesma instrução pode ser feita com ponteiros, considerando que o nome do vetor é na verdade um ponteiro para o primeiro elemento do vetor:

```
*(table+2)
```

A expressão (table+2) resulta no endereço do elemento de índice 2 da matriz. Se cada elemento da matriz é um inteiro (2 bytes), então vão ser pulados 4 bytes do início do endereço da matriz para atingir o elemento de índice 2.

Em outras palavras, a expressão (table+2) não significa avançar 2 bytes além de table e sim 2 elementos da matriz: 2 inteiros se a matriz for inteira, 2 floats se a matriz for float e assim por diante. Ou seja,

```
*(matriz + indice) == matriz[indice]
```

Existem duas maneiras de referenciar o endereço de um elemento da matriz: em notação de ponteiros, `matriz+indice`, ou em notação de matriz, `&matriz[indice]`.

Vamos mostrar um exemplo para esclarecer estes conceitos:

```
/* media de um numero arbitrario de notas */
/* exemplo de ponteiros */
#include <stdio.h>
#define LIM 40

void main()
{
    float notas[LIM], soma = 0.0;
    int register i = 0;
    do
    {
        printf("Digite a nota do aluno %d: ", i);
        scanf("%f", notas+i);
        if(*(notas+i) > 0)
            soma += *(notas+i);
    }while(*(notas+i++) > 0);
    printf("Media das notas: %.2f", soma/(i-1));
}
```

O operador de incremento ou decremento não pode ser aplicado a ponteiros constantes. Por exemplo, não podemos substituir o comando (notas + i++) por (notas++), haja visto que notas é um ponteiro constante.

Para fazer isto, precisamos criar um ponteiro variável qualquer, atribuir a ele o endereço da matriz e depois incrementar o valor deste ponteiro:

```
float * ptr = 0;
```

```
ptr = notas;  
(notas + i++) == (ptr++) /* são equivalentes */
```

Como, ptr apontar para uma matriz do tipo float, o operador (++) incrementa em 4 bytes a cada operação.

Podemos passar matrizes como parâmetros para funções, passando-se como parâmetro um ponteiro para o primeiro elemento da matriz e depois utilizando este ponteiro para acessar todos os elementos da matriz. Esta sintaxe funciona exatamente igual como se estivessemos passando um ponteiro de uma variável normal como parâmetro. Veja a secção anterior para verificar esta sintaxe.

11.5 Ponteiros & Strings

Como strings são na verdade tratadas em C como matrizes simples de caracteres finalizadas por '0' (caracter '\0'), podemos utilizar as facilidades fornecidas pelos ponteiros para manipular strings também. Abaixo mostramos um exemplo que ilustra este caso:

```
/* procura um caractere em uma string */  
#include <stdio.h>  
#include <conio.h>  
char * procstr(char *, char);  
  
void main()  
{  
    char *ptr;  
    char ch, lin[81];  
    puts("Digite uma sentença: ");  
    gets(lin);  
    printf("Digite o caractere a ser procurado: ");  
    ch=getche();  
    ptr = procstr(lin, ch);  
    printf("\nA string começa no endereço %p.\n", lin);  
    if(ptr) /* if(ptr != 0) */  
    {  
        printf("Primeira ocorrência do caractere: %p.\n", ptr);  
        printf("E a posição: %d", ptr-lin);  
    }  
    else  
        printf("\nCaractere não existe.\n");  
}  
  
char * procstr(char * l, char c)  
{  
    if(l == 0)  
        return 0;  
    while((*l != c)&&(*l != '\0'))  
        l++;  
    if(*l != '\0')  
        return l;  
    return 0;  
}
```

Primeiramente, fornecemos aqui um exemplo de função que retorna como resultado um ponteiro. Note que, antes de acessarmos o valor deste ponteiro, testamos se este é um ponteiro válido para não cometer nenhum erro acessando uma área imprópria da memória.

O programa utilizou a manipulação de ponteiros para localizar a posição de um caracter em uma string e fornecer a sua posição ao usuário. A maioria das funções em C manipulam strings como ponteiros e, por isso, são extremamente rápidas e otimizadas.

Em vez de declararmos uma string como uma tabela, podemos fazê-lo diretamente como sendo um ponteiro. O tipo (char *) é reconhecido em C como sendo um tipo string, ou seja, um ponteiro para uma sequência de caracteres. Por isso, as inicializações abaixo são equivalentes:

```
char * salute = "Saudacoes";  
char salute[] = "Saudacoes";
```

Entretanto, esta inicialização alocará memória somente para o correspondente número de caracteres da string passada mais um caracter '\0' para delimitar a string. Nenhum espaço a mais de memória será alocado neste caso.

As duas formas provocam o mesmo efeito, mas são diferentes: a primeira declara salute como um ponteiro variável e a segunda como um ponteiro constante. O valor de um ponteiro constante não pode ser modificado, já um ponteiro variável pode ter seu valor modificado, por um incremento (++), por exemplo.

Podemos criar em C matrizes de ponteiros para strings. Esta é uma das maneiras mais econômicas para alocar memória para uma matriz de strings sem desperdiçar memória. Por exemplo, a inicialização:

```
static char * list[5] =  
{  
    "Katarina",  
    "Nigel",  
    "Gustavo",  
    "Francisco",  
    "Airton"      };
```

Vai alocar uma matriz com 5 elementos, onde cada elemento conterá o valor de um ponteiro para uma lista de caracteres (string). Desta forma, o espaço de memória necessário para alocar será somente o tamanho estritamente necessário. Como a string é de ponteiros, basta alocar espaço para cada elemento de 2 bytes, enquanto que a sequência de caracteres apontado por este ponteiro pode estar em qualquer posição de memória.

Se quiséssemos fazer a mesma inicialização utilizando uma matriz de vetores de caracteres, isto implicaria que teríamos que alocar uma tabela MxN para conter todos os caracteres e isto nos levaria a desperdiçar algumas posições da memória que não chegaram nunca a conter algum caracter, já que as strings se caracterizam por conter um comprimento diferentes de caracteres.

Por isto, uma das razões para se inicializar "strings" com ponteiros é a alocação mais eficiente de memória. Uma outra razão é a de obter maior flexibilidade para manipular matrizes de strings.

Por exemplo, suponha que desejássemos reordenar as strings da matriz acima. Para fazer isto, não precisamos remover as strings de sua posição e escrevê-las em outra matriz na ordem correta, basta trocar de posição os ponteiros que apontam para as strings. Reordenando os ponteiros, obteremos a ordem desejada das strings sem ter que se preocupar em reescrever as strings em outra posição de memória.

11.6 Ponteiros para Ponteiros

A habilidade da linguagem C de tratar partes de matrizes como matrizes cria um novo tópico de C, ponteiros que apontam para ponteiros. Esta habilidade dá a C uma grande flexibilidade na criação e ordenação de dados complexos.

Vamos analisar um exemplo de acesso duplamente indireto de dados derivados de uma matriz de duas dimensões.

```
/* uso de ponteiros para ponteiros */
#include <stdio.h>
#define LIN 4
#define COL 5

void main()
{
    static int tabela[LIN][COL] =
    {
        {13, 15, 17, 19, 21},
        {20, 22, 24, 26, 28},
        {31, 33, 35, 37, 39},
        {40, 42, 44, 46, 48} };
    int c = 10;
    int j, k;
    int * ptr = ( int *) tabela;
    for(j=0; j<LIN; j++)
        for(k=0; k<COL; k++)
            *(ptr + j*COL + k) += c;
    for(j=0; j<LIN; j++)
    {
        for(k=0; k<COL; k++)
            printf("%d ", (*(tabela+j)+k));
        printf("\n");
    }
}
```

Neste exemplo, estamos utilizando o ponteiro tabela para acessar os termos da matriz bidimensional. Mas como fazer para acessar um termo posicionado em `tabela[i][j]`?

Como tabela é uma matriz para inteiros, cada elemento ocupará dois bytes e cada coluna com 5 elementos ocupará 10 bytes. Abaixo, mostramos o armazenamento desta tabela na memória (Os endereços estão em números decimais para facilitar o entendimento).

Tabela:	i\j	0	1	2	3	4
Mem\offset	i\offset	0	2	4	6	8
1000	0	13	15	17	19	21
1010	1	20	22	24	26	28
1020	2	31	33	35	37	39
1030	3	40	42	44	46	48

Vamos tentar agora acessar o elemento `tabela[2][4]`, onde a tabela tem dimensão 4x5.

a) `*(tabela + 2*5 + 4)`

Note que, podemos tratar uma matriz $m \times n$ como sendo um vetor simples de inteiros, pois as linhas são posicionadas na memória uma após a outra. Então, se calcularmos a posição de memória ocupada por `tabela[2][4]`, poderemos acessar o seu valor através de um ponteiro:

- `int * ptr = tabela` : contém o endereço inicial da matriz, no caso 1000.
- `(ptr + 2*5) == (ptr + 10)` : contém a posição inicial da terceira linha da tabela (`tabela[2]`), no caso 1020 pois cada elemento ocupa 2 bytes (int). Isto fica claro na matriz acima. Sabendo-se que a terceira linha começa após 2 linhas com 5 elementos, basta adicionar o número de termos contidos nestas linhas ao ponteiro da matriz e obtemos um ponteiro para o início da linha 2.

- $((ptr + 2*5) + 4) == (ptr + 14)$: contém a posição de memória onde está o termo `tabela[2][4]`, no caso 1028. Com o ponteiro da linha, queremos agora acessar o termo 4 desta linha. Então, basta adicionar 4 ao ponteiro da linha, que obtemos um ponteiro para o referido termo.
- $*(ptr + 2*5 + 4) == *(ptr + 14)$: calculada a posição de memória onde está o referido termo, basta utilizar o operador `(*)` para acessá-lo.

b) $*(*(tabela + 2) + 4)$

Considere a declaração da seguinte tabela 4x5:

```
int tabela[4][5];
```

Queremos agora criar um vetor que contenha a terceira linha desta tabela. Podemos fazer isto, usando uma das duas declarações equivalentes:

```
int linha[5] = tabela[2];  
int linha[5] = *(tabela + 2); /* eq. 1 */
```

Note que, uma matriz é na verdade um vetor contendo outras matrizes de ordem menor. Na declaração acima, fazemos a atribuição de um elemento da matriz `tabela`, isto é, um vetor com 5 inteiros, para uma matriz que contém 5 inteiros. A diferença das notações é que a segunda utiliza a manipulação de ponteiros na sua declaração.

Agora, vamos acessar o quinto termo de linha, isto é, `linha[4]`. Podemos fazer isto pelo método convencional de matrizes ou por ponteiros. Os dois métodos abaixo são equivalentes:

```
int num = linha[4];  
int num = *(linha + 4);
```

Desta forma, conseguimos acessar indiretamente o termo `tabela[2][4]`. Primeiro atribuímos a terceira linha da tabela a um vetor simples e depois acessamos o quinto elemento deste vetor. Entretanto, C nos permite realizar o mesmo método de acesso, sem criar este passo intermediário, basta fazer:

```
int tabela[4][5];  
int num = (*(tabela + 2) + 4); /* tabela[2][4]; */
```

Assim, $*(tabela+2)$ retorna o endereço da terceira linha da matriz `tabela`, equivalente a escrevermos `&tabela[2][0]`. Ao adicionarmos 4 a este endereço, calculamos o endereço do quinto elemento nesta linha.

Portanto, o endereço do quinto elemento é $*(tabela+2)+4$ e o conteúdo deste endereço é $*(*(tabela+2)+4)$, que é 39. Esta expressão é m ponteiro para ponteiro. Este princípio é aplicado para matrizes de qualquer dimensão, em outras palavras:

```
tabela[j][k] = (*(tabela + j) + k);  
cubo[i][j][k] = (*( *(cubo + i) + j) + k);
```

Desta maneira, C nos permite criar ponteiros para ponteiros e fazer estruturas tão complexas e flexíveis quanto desejarmos. O operador `*(ptr)` pode então ser aninhado para obter o valor final apontado pela sequência de ponteiros. Esta técnica pode fornecer grande velocidade de execução e economia de memória, mas tenha cuidado para construir um código claro de forma a evitar problemas devido a um gerenciamento ruim destes ponteiros.

11.7 Argumentos da Linha de Comando

C permite que um programa receba uma lista de parâmetros através da função `main()`. A forma geral da função `main` é dada por:

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

onde `argc` é o número de argumentos passados para a função, e `argv` é uma matriz de string que contém todos os argumentos passados. A função `main()` nesta forma geral necessita retornar um valor indicando o resultado do programa ao sistema operacional. No caso, o retorno do valor 0 indicará que o programa foi executado adequadamente.

Suponha que você tenha criado um programa chamado `jogo` e tenha executado ele no sistema operacional com a linha de comando:

```
C:> jogo xadrex.txt 2 3.14159
```

Ao executar o programa passo-a-passo, verificaremos que `argc` será igual a 4, indicando que o usuário forneceu 4 parâmetros ao programa. Sendo que os parâmetros estão armazenados em `argv` na forma de string e conterão os seguintes valores:

```
argv[0] == "jogo";  
argv[1] == "xadrex.txt";  
argv[2] == "2";  
argv[3] == "3.14159";
```

Conhecendo como funciona os parâmetros da função `main()`, você já pode utilizar a atribuição abaixo para obter os valores fornecidos, bastando converter o parâmetro do formato ASCII para o tipo de dado requerido :

```
char * param = argv[3];
```

11.8 Ponteiros para Estruturas

Como já mostramos ponteiros são mais fáceis de manipular que matrizes em diversas situações, assim ponteiros para estruturas são mais fáceis de manipular que matrizes de estruturas. Várias representações de dados que parecem fantásticas são constituídas de estruturas contendo ponteiros para outras estruturas. O nosso próximo exemplo mostra como definir um ponteiro para estrutura e usá-lo para acessar os membros da estrutura.

```
/* mostra ponteiro para estrutura */  
#include <stdio.h>  
  
struct lista /* declara estrutura */  
{  
    char titulo[30];  
    char autor[30];  
    int regnum;  
    double preco;  
};  
  
int main(int argc, char * argv[])  
{  
    static struct lista livro[2] =  
    {  
        { "Helena", "Machado de Assis", 102, 70.50 },  
        { "Iracema", "Jose de Alencar", 321, 63.25 }  
    };  
    struct lista *ptr1 = 0; /* ponteiro para estrutura */
```

```
printf("Endereco #1: %p      #2: %p\n", &livro[0], &livro[1]);
ptr1 = &livro[0];
printf("Ponteiro #1: %p      #2: %p\n", ptr1, ptr1 + 1);
printf("ptr1->preco: R$%.2f \t (*ptr1).preco:
      R$%.2f\n", ptr1->preco, (*ptr1).preco);
ptr1++; /* Aponta para a proxima estrutura */
printf("ptr1->titulo: %s \t ptr1->autor: %s\n",
      ptr1->titulo, ptr1->autor);

return 0;
}
```

A declaração é feita como se estivéssemos declarando uma variável de qualquer tipo, adicionando-se o operador (*) na frente do nome da variável. Por isso, a declaração de um ponteiro para uma estrutura é feita na forma :

```
struct lista *ptr1;
```

O ponteiro ptr1 pode então apontar para qualquer estrutura do tipo lista.

A atribuição de um endereço a um ponteiro de estrutura funciona da mesma forma como uma variável qualquer, empregando-se o operador (&):

```
ptr1 = &(livro[0]);
```

Vimos no capítulo sobre estruturas como fazer o acesso a um elemento de uma estrutura através do operador (.). Por exemplo, se quisermos ler o valor do preco da primeira estrutura da matriz livro, procederíamos da forma:

```
livro[0].preco = 89.95;
```

Mas como proceder com ponteiros? Podemos fazê-lo de duas formas.

Primeiro, podemos utilizar o operador (*) para obter a estrutura apontada por um ponteiro e depois empregar o operador normal (.) para acessar um elemento desta estrutura. Aplicando no exemplo acima, teremos:

```
(*ptr1).preco = 89.95;
```

O segundo método utiliza o operador (->) que nos permite acessar um elemento de uma estrutura apontada por um dado ponteiro. Aplicando-se este operador no problema acima, temos:

```
ptr1->preco = 89.95;
```

Em outras palavras, um ponteiro para estrutura seguido pelo operador (->) trabalha da mesma maneira que o nome de uma estrutura seguido pelo operador (.). É importante notar que ptr1 é um ponteiro, mas ptr1->preco é um membro da estrutura apontada. Neste caso, ptr1->preco é uma variável double.

O operador (.) conecta a estrutura a um membro dela; o operador (->) conecta um ponteiro a um membro da estrutura.

11.9 Alocação Dinâmica de Memória

A linguagem C oferece um conjunto de funções que permitem a alocação ou liberação dinâmica de memória. Desta forma, podemos alocar memória para um programa de acordo com a sua necessidade instantânea de memória.

A memória de trabalho do computador (RAM) usualmente é subdividida em vários segmentos lógicos dentro de um programa. Estes segmentos são:

- segmento de **dados**, onde são alocadas as variáveis globais (extern), definidas em pre-run-time;
- o segmento de **código**, onde estão as instruções de máquina do programa em si;
- o segmento de **pilha** (“stack”), onde as funções alocam provisoriamente suas variáveis locais (auto). Este segmento também é usado para passagem de parâmetros;
- o segmento **extra**, que pode conter mais variáveis globais;

Toda a área de memória restante entre o fim do programa e o fim da RAM livre é chamada de **heap**. O **heap** é usado para a criação de variáveis dinâmicas, que são criadas em **run-time** (isto é, durante a execução do programa). Este tipo de variável é útil quando não se sabe de antemão quantas variáveis de um determinado tipo serão necessárias para a aplicação em questão.

Quando escrevemos um programa utilizando o método de declaração de variáveis visto anteriormente (alocação estática de memória), o programa ao ser executado alocará somente um bloco fixo de memória para armazenar todos os seus dados. Isto resolve o problema de termos um espaço de memória alocado para podermos armazenar os dados do programa. Entretanto, como visto no capítulo de matrizes, este método não otimiza a utilização do espaço de memória alocado. Por exemplo, imagine que você precise de uma matriz temporária para armazenar alguns dados temporários durante a execução de uma dada função de manipulação de matrizes. Para tal, você deverá declarar esta matriz na função, o que implicará que o computador irá alocar um bloco de memória para esta matriz. Este espaço de memória ficará alocado ao seu programa durante toda a execução deste, apesar do programa só utilizar uma vez esta matriz e, posteriormente, não precisar mais desta matriz e nem do espaço de memória alocado a esta.

Com a alocação dinâmica de memória, podemos, em tempo de execução, fazer com que um programa alogue um determinado espaço de memória, utilize este espaço por um determinado tempo e depois o libere, para que outros programas possam vir a utilizá-lo. No caso do nosso exemplo, podemos fazer com que sempre que a função for chamada, ela alocará um espaço de memória para armazenar a referida matriz e após o seu uso, o programa liberará este bloco de memória para que outro programa o utilize. Desta forma, se executarmos esta função apenas uma vez, o programa irá liberar esta memória posteriormente, permitindo assim que outros programas façam um uso mais adequado desta.

Desta forma, a alocação dinâmica de memória é utilizada em programas para alocar e liberar blocos temporários de memórias durante a execução de um programa (por isso é chamado alocação dinâmica). Este bloco de memória é solicitado ao sistema operacional que procura um espaço livre de memória para o programa. Se o sistema operacional achar um bloco de memória livre do tamanho do bloco solicitado, este passa o bloco de memória para o controle do programa e não irá permitir que nenhum outro programa utilize esta memória enquanto ela estiver alocada. No final do seu uso, o programa libera novamente esta memória ao sistema operacional.

Outro exemplo de aplicação da alocação dinâmica de memória é na utilização de matrizes quando não sabemos de antemão quantos elementos serão necessários. Desta forma, podemos utilizar a alocação dinâmica de memória para somente alocar a quantidade necessária de memória e no momento em que esta memória for requerida.

11.9.1 Malloc()

A função `malloc()` é utilizada para fazer a alocação dinâmica de um bloco de memória a um dado programa. A função `malloc()` toma um inteiro sem sinal como argumento. Este número representa a quantidade em bytes de memória requerida. **A função retorna um ponteiro para o primeiro byte do novo bloco de memória que foi alocado.**

É importante verificar que o ponteiro retornado por `malloc()` é para um tipo `void`. O conceito de ponteiro para `void` deve ser introduzido para tratar com situações em que seja

necessário que uma função retorne um ponteiro genérico, i.e., que possa ser convertido em um ponteiro para qualquer outro tipo de dado. Este ponteiro void pode ser convertido para um ponteiro do tipo de dado desejado (int, float, struct, ...) empregando-se o método de conversão de tipos apresentado na secção sobre tipos de dados (ver 2.5). No próximo exemplo, mostraremos o seu emprego novamente.

Quando a função `malloc()` não encontrar espaço suficiente de memória para ser alocado, esta retornará um ponteiro NULL, i.e., um ponteiro inválido.

O exemplo abaixo mostra como a função `malloc()` opera. Este programa declara uma estrutura chamada `xx` e chama `malloc()` 4 vezes. A cada chamada, `malloc()` retorna um ponteiro para uma área de memória suficiente para guardar um nova estrutura.

```
/* testa malloc() */
#include <stdio.h>

struct xx
{
    int num1;
    char chl;
};

void main()
{
    struct xx *ptr = 0;
    int j;
    printf("sizeof(struct xx) = %d\n", sizeof(struct xx));
    for(j=0; j<4; j++)
    {
        ptr = (struct xx *) malloc(sizeof(struct xx));
        printf("ptr = %x\n", ptr);
    }
}
```

Note que em nenhuma parte do programa declaramos qualquer variável estrutura. De fato, a variável estrutura é criada pela função `malloc()`; o programa não conhece o nome desta variável; mas sabe onde ela está na memória, pois `malloc()` retorna um ponteiro para ela. As variáveis criadas podem ser acessadas usando ponteiros, exatamente como se tivessem sido declaradas no início do programa.

A cada chamada de `malloc()` devemos informá-la do tamanho da estrutura que queremos guardar. Nós podemos conhecer este tamanho adicionando os bytes usados por cada membro da estrutura, ou através do uso de um novo operador em C unário chamado `sizeof()`. Este operador produz um inteiro igual ao tamanho, em bytes, da variável ou do tipo de dado que está em seu operando. Por exemplo, a expressão

`sizeof(float)`

retornará o valor 4, haja vista que um float ocupa 4 bytes. No programa exemplo, usamos `sizeof()` em `printf()` e ele retorna 3, pois a estrutura `xx` consiste em um caracter e um inteiro.

Então, `sizeof()` forneceu o tamanho em bytes da estrutura para que `malloc()` pudesse alocar o espaço de memória requerido. Feito isto, `malloc()` retornou um ponteiro do tipo void, que foi convertido para o tipo `struct xx` através da expressão:

`(struct xx *)`

11.9.2 Calloc()

Uma outra opção para a alocação de memória é o uso da função `calloc()`. Há uma grande semelhança entre `calloc()` e `malloc()` que também retorna um ponteiro para void apontando para o primeiro byte do bloco solicitado.

A nova função aceita dois argumentos do tipo `unsigned int`. Um uso típico é mostrado abaixo:

```
long * memnova;  
memnova = (long *) calloc(100, sizeof(long));
```

O primeiro argumento é o número de células de memórias desejadas e o segundo argumento é o tamanho de cada célula em bytes. No exemplo acima, `long` usa quatro bytes, então esta instrução alocará espaço para 100 unidades de quatro bytes, ou seja, 400 bytes.

A função `calloc()` tem mais uma característica: ela inicializa todo o conteúdo do bloco com zero.

11.9.3 Free()

A função `free()` libera a memória alocada por `malloc()` e `calloc()`. Aceita, como argumento, um ponteiro para uma área de memória previamente alocada e então libera esta área para uma possível utilização futura.

Sempre que um espaço de memória for alocado, este deve ser necessariamente liberado após o seu uso. Se não for liberada, esta memória ficará indisponível para o uso pelo sistema operacional para outros aplicativos. A utilização consciente da alocação e liberação dinâmica de memória, permite um uso otimizado da memória disponível no computador.

A função `free()` declara o seu argumento como um ponteiro para void. A vantagem desta declaração é que ela permite que a chamada à função seja feita com um argumento ponteiro para qualquer tipo de dado.

```
long * memnova;  
memnova = (long *)calloc(100, sizeof(long));  
/* usa memnova */  
free(memnova); /* libera a memória alocada */
```

11.10 Exercícios

11.1 Escreva um programa que receba duas strings como argumentos e troque o conteúdo de `string1` como `string2`.

11.2 Escreva um programa que inverta a ordem dos caracteres de uma string. Por exemplo, se a string recebida é “Saudacoes” deve ser modificada para “seocaduaS”.

11.3 Reescreva o programa do exercício 9.1 utilizando alocação dinâmica de memória.

11.4 Reescreva o programa do exercício 9.3 utilizando alocação dinâmica de memória.

11.5 A lista encadeada se assemelha a uma corrente em que as estruturas estão penduradas sequencialmente. Isto é, a corrente é acessada através de um ponteiro para a primeira estrutura, chamada cabeça, e cada estrutura contém um ponteiro para a sua sucessora, e o ponteiro da última estrutura tem valor NULL (0) indicando o fim da lista. Normalmente uma lista encadeada é criada

dinamicamente na memória. Crie um programa com uma lista encadeada para armazenar dados de livros em uma biblioteca.

11.6 Crie um programa com lista encadeada para catalogar clientes e fitas em uma vídeo locadora.

11.7 Crie uma estrutura para descrever restaurantes. Os membros devem armazenar o nome, o endereço, o preço médio e o tipo de comida. Crie uma lista ligada que apresente os restaurantes de um certo tipo de comida indexados pelo preço. O menor preço deve ser o primeiro da lista. Escreva um programa que peça o tipo de comida e imprima os restaurantes que oferecem este tipo de comida.

11.8 Escreva um programa para montar uma matriz de estruturas para armazenar as notas de 40 alunos. A primeira coluna da matriz deve conter o nome do aluno, a segunda o telefone, a terceira a data de nascimento, depois seguem as notas em lista e na última coluna deve ser calculada a média até o presente momento. A professora deve ser capaz de inserir e retirar alunos, e poder editar os dados dos alunos. A professora deve poder também listar os dados de todos alunos na forma de uma tabela na tela do computador. A lista de alunos deve ser indexada pelo nome destes. Utilize a idéia da lista ligada e da alocação dinâmica de memória.

12 Manipulação de Arquivos em C

Neste capítulo, veremos brevemente a manipulação de arquivos em C. Em um capítulo posterior, será apresentada novamente a manipulação de arquivos agora utilizando C++.

12.1 Tipos de Arquivos

Uma maneira de classificar operações de acesso a arquivos é conforme a forma como eles são abertos: em modo texto ou em modo binário.

Arquivos em modo texto, operam em dados armazenados em formato texto, ou seja, os dados são traduzidos para caracteres e estes caracteres são escritos nos arquivos. Por esta razão, fica mais fácil de compreender os seus formatos e localizar possíveis erros.

Arquivos em modo binário, operam em dados binários, ou seja, os dados escritos neste formato são escritos na forma binária, não necessitando de nenhuma conversão do tipo do dado utilizado para ASCII e ocupando bem menos memória de disco (arquivos menores).

Uma outra diferença entre o modo texto e o modo binário é a forma usada para guardar números no disco. Na forma de texto, os números são guardados como cadeias de caracteres, enquanto que na forma binária são guardados com estão na memória, dois bytes para um inteiro, quatro bytes para float e assim por diante.

12.2 Declaração, abertura e fechamento

C declara um tipo especial de estrutura, chamada FILE, para operar com arquivos. Este tipo é definido na biblioteca “stdio.h”, que deve ser incluída na compilação com a diretiva #include para permitir operações sobre arquivos. Os membros da estrutura FILE contêm informações sobre o arquivo a ser usado, tais como: seu atual tamanho, a localização de seus buffers de dados, se o arquivo está sendo lido ou gravado, etc.

Toda operação realizada sobre um arquivo (abertura, fechamento, leitura, escrita) requer um apontador para uma estrutura do tipo FILE:

```
FILE *File_ptr;
```

A abertura de um arquivo é feita com a função `fopen()`:

```
File_ptr = fopen("Nome do Arquivo", "<I/O mode>");
```

onde as opções para a abertura do arquivo estão listadas abaixo:

I/O mode	Função:
r	Read, abre arquivo para leitura. O arquivo deve existir.
w	Write, abre arquivo para escrita. Se o arquivo estiver presente ele será destruído e reinicializado. Se não existir, ele será criado.
a	Append, abre arquivo para escrita. Os dados serão adicionados ao fim do arquivo se este existir, ou um novo arquivo será criado.
r+	Read, abre um arquivo para leitura e gravação. O arquivo deve existir e pode ser atualizado.
w+	Write, abre um arquivo para leitura e gravação. Se o arquivo estiver presente ele será destruído e reinicializado. Se não existir, ele será criado.
a+	Append, abre um arquivo para atualizações ou para adicionar dados ao seu final.
t	Text, arquivo contém texto dados em ASCII.
b	Binary, arquivo contém dados em binário.

A função `fopen()` executa duas tarefas. Primeiro, ela preenche a estrutura `FILE` com as informações necessárias para o programa e para o sistema operacional, assim eles podem se comunicar. Segundo, `fopen()` retorna um ponteiro do tipo `FILE` que aponta para a localização na memória da estrutura `FILE`.

A função `fopen()` pode não conseguir abrir um arquivo por algum motivo (falta de espaço em disco, arquivo inexistente, etc.) e por isso esta retornará um ponteiro inválido, isto é, contendo o valor `NULL (0)`. Por isso, teste sempre se o ponteiro fornecido por `fopen()` é válido antes de utilizado, caso contrário o seu programa pode vir a ter uma falha séria.

Quando terminamos a gravação do arquivo, precisamos fechá-lo. O fechamento de um arquivo é feito com a função `fclose()`:

```
fclose(File_ptr);
```

Quando fechamos um arquivo é que o sistema operacional irá salvar as suas modificações ou até mesmo criar o arquivo, no caso de ser um arquivo novo. Até então, o sistema operacional estava salvando as alterações em um buffer antes de escrever estes dados no arquivo. Este procedimento é executado para otimizar o tempo de acesso ao disco empregado pelo sistema operacional. Por isso, não esqueça de fechar um arquivo, senão os seus dados podem ser perdidos e o arquivo não seja criado adequadamente.

Uma outra razão para fechar o arquivo é a deliberação das áreas de comunicação usadas, para que estejam disponíveis a outros arquivos. Estas áreas incluem a estrutura `FILE` e o buffer.

Uma outra função que fecha arquivos é a função `exit()`. A função `exit()` difere da função `fclose()` em vários pontos. Primeiro, `exit()` fecha todos os arquivos abertos. Segundo, a função `exit()` também termina o programa e devolve o controle ao sistema operacional.

A função `fclose()` simplesmente fecha o arquivo associado ao ponteiro `FILE` usado como argumento.

12.3 Leitura e escrita de caracteres

A função usada para ler um único caracter de um arquivo é `getc()` enquanto que a função `putc()` escreve um caracter em um arquivo. Abaixo, apresentamos exemplos destas funções:

Escrita	Leitura
<pre>#include <stdio.h> FILE *fileptr; char filename[65]; char mychar; fileptr = fopen(filename, "w"); putc(mychar, fileptr); fclose(fileptr);</pre>	<pre>#include <stdio.h> FILE *fileptr; char filename[65]; int mychar; int i = 0; fileptr = fopen(filename, "r"); mychar = getchar(fileptr); while(mychar != EOF) { printf("%c", mychar); mychar = getchar(fileptr); } fclose(fileptr);</pre>

12.4 Fim de Arquivo (EOF)

EOF é um sinal enviado pelo sistema operacional para indicar o fim de um arquivo. O sinal EOF (Fim de Arquivo) enviado pelo sistema operacional para o programa C não é um caracter, e sim um inteiro de valor `-1` e está definido em `stdio.h`.

Perceba que no exemplo anterior de leitura de caracteres, nós usamos uma variável inteira para guardar os caracteres lidos para que possamos interpretar o sinal de EOF. Se usarmos uma variável do tipo char, o caractere de código ASCII 255 decimal (0xFF em Hexa) será interpretado como EOF. Queremos usar todos os caracteres de 0 a 255 em nosso arquivo e uma variável inteira nos assegura isto. Neste exemplo, EOF é usado para ler todos os caracteres de um dado arquivo, quando não conhecemos de antemão a quantidade de caracteres deste arquivo.

A marca de fim de arquivo pode ser diferente para diferentes sistemas operacionais. Assim, o valor de EOF pode ser qualquer. O seu arquivo stdio.h define EOF com o valor correto para o seu sistema operacional; assim, em seus programas, use EOF para testar fim de arquivo.

O fim de um arquivo pode também ser determinado utilizando-se a função `fEOF()`, que recebe como parâmetro um ponteiro válido para a estrutura FILE.

12.5 Leitura e escrita de strings

A função `fputs()` escreve uma string em um arquivo e, por isso, toma dois argumentos, sendo o primeiro a matriz de caracteres que será gravada e o segundo o ponteiro para a estrutura FILE do arquivo a ser gravado. Observe que a função `fputs()` não coloca automaticamente o caractere de nova-linha no fim de cada linha. No programa exemplo abaixo, fazemos isto explicitamente com o caractere `'\n'`.

A função `gets()` lê uma linha por vez de um arquivo texto. A função `gets()` toma 3 argumentos. O primeiro é um ponteiro para o buffer onde será colocada a linha lida. O segundo é um número inteiro que indica o limite máximo de caracteres a serem lidos. Na verdade, este número deve ser pelo menos um maior que o número de caracteres lidos, pois `gets()` acrescenta o caractere NULL (`'\0'`) na próxima posição livre. O terceiro argumento é um ponteiro para a estrutura FILE do arquivo a ser lido. A função termina a leitura após ler um caractere de nova linha (`'\n'`) ou um caractere de fim de arquivo (EOF).

Escrita:

```
#include <stdio.h>
FILE *fileptr;
char filename[65];
char line[81];
fileptr = fopen(filename, "w");
fputs(line, fileptr); /* fprintf(fileptr, "%s\n", line) */
fputs("\n", fileptr); /* pode ser usado aqui no lugar */
fclose(fileptr);      /* dos dois fputs */
```

Leitura:

```
#include <stdio.h>
FILE *fileptr;
char filename[65];
char line[81];
fileptr = fopen(filename, "r");
fgets(line, 80, fileptr); /* fscanf(fileptr, "%s", line); */
close(fileptr);          /* pode ser usado no lugar de fgets */
```

12.6 Arquivos Padrões

C define um conjunto de arquivos padrões utilizados para acessar alguns periféricos do computador (como a impressora) ou para ler da entrada padrão (normalmente o teclado) ou escrever para a saída padrão (normalmente a tela).

Desta forma, `_streams[]` foi criada como uma matriz de estruturas `FILE`. Se você perder um tempo e analisar o seu arquivo `stdio.h`, encontrará várias constantes simbólicas definidas como:

```
#define stdin      (&_streams[0])
#define stdout     (&_streams[1])
#define stderr     (&_streams[2])
#define stdaux     (&_streams[3])
#define stdprn     (&_streams[4])
```

Estas constantes podem ser usadas para acessar qualquer um dos 5 arquivos padrão que são predefinidos pelo MS-DOS e abertos automaticamente quando o seu programa inicia a sua execução e fechados ao seu fim.

Nome:	Periférico:
stdin	Standard input device (teclado)
stdout	Standard output device (tela)
stderr	Standard error device (tela)
stdaux	Standard auxiliary device (porta serial)
stdprn	Standard printing device (impressora paralela)

Cada uma destas constantes pode ser tratada como um ponteiro para uma estrutura FILE dos arquivos in, out, err, aux e prn respectivamente. Você pode usar os ponteiros FILE definidos em stdio.h para acessar os periféricos predefinidos pelo MS-DOS ou usar seus nomes e definir os ponteiros necessários. Como exemplo, a instrução:

```
fgets(string, 80, stdin);
```

lê uma string do teclado.

A instrução:

```
fputs(string, stdprn);
```

imprimirá uma string na impressora.

12.7 Gravando um Arquivo de Forma Formatada

Nos capítulos iniciais apresentamos a função `printf()` para imprimir na tela dados de forma formatada. Para realizar a mesma tarefa, entretanto não para escrever na tela mas sim para um arquivo, foi criada a função `fprintf()`. Esta função é similar a `printf()` exceto que o ponteiro para `FILE` é tomado como primeiro argumento. Como em `printf()`, podemos formatar os dados de várias maneiras; todas as possibilidades de formato de `printf()` operam com `fprintf()`.

Da mesma forma foi criada a função `fscanf()`, que como `scanf()`, lê um dado formatado. A diferença consiste que `fscanf()` lê um dado de um arquivo e recebe um ponteiro para `FILE` como primeiro argumento.

Exemplo :

```
#include <stdio.h>
FILE *fptr;
int size = 0;
fptr = fopen("dados.txt", "rw");
fscanf(fptr, "%d", &size);
fprintf(fptr, "%s %d %f", "Casa Nova", 12, 13.45);
fclose(fptr);
```

12.8 Leitura e escrita de valores binários

Quando desejamos operar com arquivos no modo binário, basta adicionar o caracter 'b' no I/O Mode da função `open()`, como apresentado anteriormente. As funções apresentadas anteriormente podem ser usadas para ler e escrever no modo binário, entretanto apresentaremos aqui duas novas funções que facilitam este processo: `fwrite()` e `fread()`. Estas funções são empregadas para escrever/ler os dados armazenados em um bloco de memória (um buffer de memória) em um arquivo. Aplicações típicas e na escrita/leitura de dados complexos como matrizes e estruturas.

A função `fwrite()` toma 4 argumentos. O primeiro é um ponteiro do tipo void que aponta para a localização na memória do dado a ser gravado. O segundo argumento é um número inteiro que indica o tamanho do tipo de dado a ser gravado. Normalmente, pode-se utilizar o operador `sizeof()` para se obter este valor. O terceiro argumento é um número inteiro que informa a `fwrite()` quantos itens do mesmo tipo serão gravados. O quarto argumento é um ponteiro para a estrutura FILE do arquivo onde queremos gravar.

A função `fread()` toma também 4 argumentos. O primeiro é um ponteiro void para a localização da memória onde serão armazenados os dados lidos. O segundo indica também a quantidade de bytes do tipo de dado a ser lido. O terceiro argumento informa a quantidade de itens a serem lidos a cada chamada, e o quarto argumento é um ponteiro para a estrutura FILE do arquivo a ser lido.

A função `fread()` retorna o número de itens lidos. Normalmente este número deve ser igual ao terceiro argumento. Se for encontrado o fim do arquivo, o número será menor que o valor do terceiro argumento, podendo ser zero caso nenhum dado tenha sido lido.

As funções `fread()` e `fwrite()` trabalham com qualquer tipo de dado, incluindo matrizes e estruturas, e armazenam números em formato binário.

Escrita:
<code>fileptr = fopen(filename, "wb");</code> <code>fwrite(&dados, sizeof(dados), 1, fileptr);</code>
Leitura:
<code>fileptr = fopen(filename, "rb");</code> <code>fread(&dados, sizeof(dados), 1, fileptr);</code>

Se "filename" for inicializado com "prn", os dados são enviados a impressora.

Exemplo:

```
fileptr=fopen(filename,"rb");
while(!feof(fileptr))
{
    fread(&dados, sizeof(dados),1,fileptr);
}
fclose(fileptr);
```

12.9 Exercícios

12.1 Escreva um programa que imprima um arquivo na tela de 20 em 20 linhas. O arquivo de entrada deve ser fornecido na linha de comando. A cada impressão de 20 linhas, o programa aguarda o pressionamento de uma tecla.

12.2 Escreva um programa que imprima o tamanho de um arquivo em bytes. O nome do arquivo deve ser fornecido na linha de comando.

12.3 Escreva um programa que criptografa um arquivo usando o operador de complemento de bit-a-bit (~). Quando o programa é executado para um arquivo já criptografado, o arquivo é recomposto e volta ao original.

12.4 Refaça o problema 11.5, agora salvando a lista de livros em um arquivo. Permita que o usuário possa retirar um livro desta lista, apagando-o do arquivo, ou adicionar um livro em uma posição determinada na lista, reescrevendo a lista no arquivo. Utilize o modo texto para a manipulação de arquivos.

12.5 Como o exemplo anterior, refaça o problema 11.6 mas agora utilizando o modo binário.

13 Programação em C++

A linguagem C foi e continua sendo uma das linguagens mais importantes na área da informática. Apesar do seu ótimo desempenho em termos de programas otimizados, rápidos e portáteis sofreu duras críticas com relação à qualidade do código gerado considerando-se outros aspectos de relevância da engenharia de software como: legibilidade do código, sua reusabilidade e facilidade de manutenção.

Um programa de computador sempre busca representar uma dada realidade (mundo real ou modelo abstrato: matemático) através de uma linguagem de programação. Quando criamos qualquer programa simples de computador, estamos na verdade criando um modelo da realidade e usando este modelo para analisar e estudar esta realidade. O nível de detalhe com que criamos um programa depende diretamente do nível de detalhe necessário para modelar o ambiente de acordo com as necessidades impostas pelo usuário.

Modelar um determinado ambiente complexo utilizando-se somente as estruturas e funcionalidades disponíveis por C (linguagem estrutural) é uma tarefa árdua. Por esta razão, alguns programadores em C não conseguem muitas vezes entender determinados códigos em C, pois não conseguem entender o modelo da realidade gerado por este programa ou a sua estrutura.

Neste sentido, criou-se uma forma totalmente nova e revolucionária para se modelar e simular um ambiente dado (mundo real ou sistema abstrato): a Orientação a Objetos. A Orientação a Objetos busca modelar um ambiente utilizando-se dos próprios elementos presentes neste ambiente, ou seja, os objetos. Todo ambiente pode ser modelado e simulado a partir de uma descrição dos objetos que o compõe e das relações entre eles.

Por exemplo, suponha que você esteja criando um programa para permitir que um arquiteto crie um modelo gráfico de uma sala-de-estar e apresente este modelo ao seu cliente. Empregando a metodologia de Orientação a Objetos, primeiramente o arquiteto terá que identificar os objetos que compõem a sala-de-estar que, considerando o nível de abstração do seu modelo, devem ser descritos. Neste caso, podemos facilmente listar alguns objetos que podem ser definidos: janela, porta, parede, sofá, mesa, quadro, tapete, vaso de flores, etc.

A quantidade de objetos e o nível de detalhe da sua descrição dependerão necessariamente do nível de abstração do modelo, ou seja, daquilo que o arquiteto considera importante que esteja no modelo. Por exemplo, representar o objeto “mesa” pode significar para um marceneiro definir a sua geometria, o tipo de material, o tipo de verniz requerido e o tipo do seu acabamento, enquanto que para um arquiteto significa definir a textura da mesa, o tom da sua cor, o seu custo, a durabilidade desta, etc. Por isso, é importante ter em mente, as necessidades do seu cliente antes de criar um modelo super detalhado e ineficiente.

Após a definição dos objetos que compõem um dado ambiente, precisamos definir as relações entre estes. Por exemplo, precisamos definir que um objeto “porta” e um objeto “janela” estão posicionados sob a mesma parede, ou seja, são propriedades de um terceiro objeto chamado “parede”. Assim, como outro objeto “parede” pode não conter uma “janela” mas um objeto do tipo “quadro”. Podemos até criar um objeto chamado “sala” que contém todos os objetos presentes na sala descrita pelo arquiteto e permitir que este arquiteto venha criar futuramente outros objetos como “cozinha” ou “quarto” e possa, assim, modelar toda uma residência.

Com este breve esclarecimento já apontamos algumas das propriedades vantajosas fornecidas pela programação Orientada a Objetos. A linguagem C++ foi criada a partir da linguagem C, acrescentando novas estruturas e mecanismos que possibilitam a geração de programas segundo a metodologia de Orientação a Objetos. Algumas linguagens como Smalltalk refletem muito mais a “cultura” ou a “metodologia” Orientada a Objetos, favorecendo a geração de programas segundo esta metodologia. C++ é uma espécie de adaptação de C a metodologia de Orientação a Objetos e, por isso, não possui todas as facilidades e mecanismos de uma linguagem

puramente Orientada a Objetos como Smalltalk. Entretanto, C++ herda de C a capacidade de gerar programas pequenos, otimizados, de “baixo-nível” e portáteis. Estes motivos propiciaram a grande difusão que a linguagem C++ vem sofrendo nos últimos anos.

A geração de códigos em C++ requer normalmente um tempo maior de desenvolvimento que um programa em C normal. Entretanto os ganhos em reusabilidade e em diminuição dos tempos de manutenção do programa, fazem com que C++ seja mais atrativo para a geração de médios e grandes sistemas. Emprega-se C basicamente quando temos que fazer um programa em pouco tempo, com grandes restrições na dimensão do código e no tempo disponível para o processamento do programa (requisitos de tempo-real).

A reusabilidade advém do fato que objetos definidos para representar um dado ambiente, podem ser empregados para representar um outro ambiente. Por exemplo, para representar o ambiente sala-de-estar, havíamos definido o objeto “mesa”. Podemos, entretanto, utilizar o mesmo objeto “mesa” para representar o ambiente cozinha, sem precisar redefini-lo novamente. Neste aspecto, ganhamos em tempo de desenvolvimento.

A modelagem orientada a objetos fornece uma estrutura bem clara para o código fonte do programa. Isto permite que outro programador possa entender o programa criado, reutilizar partes em outros programas e, ainda, facilmente localizar erros no código fonte. Por exemplo, suponha que na hora em que o programa estiver desenhando o objeto “mesa” na tela para o cliente do arquiteto apareça algum defeito na mesa. Poderemos, então, facilmente concluir que: ou o objeto mesa está mal representado (erro dentro da definição do objeto mesa), ou a relação deste objeto com os demais está mal definida (por exemplo, a posição deste objeto na sala), ou o mecanismo responsável pelo desenho deste objeto na tela não está operando corretamente. Isto nos permite que, rapidamente, possamos localizar, isolar e, em seguida, corrigir este erro.

A programação orientada a objeto se baseia no encapsulamento de uma estrutura de dados com as próprias rotinas de tratamento dos mesmos e na capacidade de herança destes dados e rotinas por outros objetos derivados.

A programação orientada a objeto traz consigo uma série de vantagens e também algumas desvantagens. Como vantagens podem-se citar:

- existem muitas ferramentas de apoio ao desenvolvimento de programas;
- os programas têm uma estrutura altamente modular, o que permite um mais fácil controle e expansão dos mesmos. Por exemplo, basta alterar as características de um objeto “mãe” para que todos os objetos “filhos” (que herdaram propriedades) sejam também automaticamente alterados de forma correspondente;
- a programação orientada a objeto se baseia fortemente na própria forma de pensar humana, ao contrário da forma algorítmica e procedural da programação convencional.

Por outro lado, há também desvantagens relativas a esta forma de programação:

- grande necessidade de memória;
- grande complexidade de gerenciamento interno das estruturas dos objetos, o que implica em velocidade de execução menor.
- difícil otimização de tempo de execução dos programas. A otimização de programas freqüentemente requer uma violação das próprias regras de programação orientada a objeto.

Estas desvantagens têm se tornado menos críticas nos últimos anos devido ao grande aumento da velocidade de processamento dos computadores bem como ao aumento de sua capacidade de memória.

Nesta etapa do trabalho, buscaremos agora apresentar os mecanismos mais importantes fornecidos pela linguagem C++, descrevendo não somente a sintaxe de C++ mas também conceitos envolvidos na programação Orientada a Objetos. Neste capítulo, apresentaremos brevemente quais são as novidades da linguagem C++. Posteriormente, apresentaremos os demais mecanismos da orientação a objetos.

13.1 Palavras-chave C++

A linguagem C++ inclui apenas 14 novas palavras reservadas às já existentes no C:

catch	inline	private	template
class	new	protected	this
delete	operator	public	virtual
friend			

13.2 Sintaxe & Variáveis

A linguagem C permite a declaração de variáveis dentro de uma função, somente antes da chamada a instruções desta função, ou seja, primeiro declaramos as variáveis e depois podemos efetuar uma dada instrução. Em C++, além destas formas pode-se declarar uma variável em qualquer ponto de um programa, inclusive entre instruções ou mesmo dentro delas.

Em C++, não é necessário utilizar a palavra typedef para se definir uma estrutura. A própria etiqueta especificando nome (estrutura ou união) já é um tipo nome. Mesmo que não tenhamos usado o typedef, o compilador C++ permitirá declarações de variáveis estruturadas sem a necessidade de struct. Assim, podemos redefinir a estrutura “list” da seguinte maneira :

```
struct list
{
    char        titulo[30];
    char        autor[30];
    int         regnum;
    double      preco;
};
list * livros; // declaração de uma variável do tipo 'list'
```

Em C++, toda variável que não for declarada explicitamente como static, será considerada do tipo extern automaticamente. Além deste default, o C++ ampliou o conceito de escopo para variáveis de tipos compostos (estruturas e classes). Essa ampliação consiste em tornar visíveis ou inacessíveis membros particulares de estruturas e classes. Abordaremos isto quando do estudo de classes.

Suponha que ao definir uma função você tenha criado uma variável local com o mesmo nome que uma variável global (extern). Ao utilizar o nome da variável, você estará acessando dentro da função somente a variável local. Entretanto, você pode utilizar o operador :: (escopo) para acessar a variável global. Veja o exemplo:

```
int n = 10; // variável extern, referenciada por: 'n' ou '::n'
void calc()
{
    int n = 5; // variável auto, local
    n = ::n;   // variável local recebe o valor da var. extern.
}
```

A linguagem C++ apresenta uma outra forma de comentários: uma linha iniciada pelo caractere de barra “/”, como indicam os seguintes exemplos:

```
// Este é um comentario em C++
// Após a barra dupla, tudo é comentário até o final da linha.
```

13.3 Laços e Comandos de Decisão

A linguagem C++ não modifica sintaxe ou comportamento dos laços e comandos condicionais do C. Porém, permite a declaração de variáveis na região de inicialização do for, ou seja, antes do laço estas variáveis inexistem. Exemplo:

```
int j =0;
for(int i =0, j=3; i+j<10; i = j++ )
```

13.4 I/O em C++: Stream

A linguagem C++ permite a utilização dos mecanismos de I/O definidos em C, mas define novas bibliotecas, mais adequadas ao paradigma da Orientação a Objetos. Essas bibliotecas constituem a chamada biblioteca Stream do C++, sendo esta biblioteca toda definida em termos de classes de objetos. Nela estão os três objetos: cin, cout e cerr destinados à entrada e saída de dados via terminal ou via arquivos. As definições e declarações necessárias para o uso de “streams” estão contidas no arquivo <iostream.h>.

13.4.1 A stream de saída cout

Cout é um objeto de uma classe de I/O predefinida em C++. Para mostrar os dados em um programa C++, deve-se utilizar a stream cout. Por default cout está associada a uma saída padrão (stdout), i.e., o ao terminal de vídeo. Sintaxe de cout :

```
cout << expressão;
onde,
<< : é o operador de inserção usado a fim de direcionar a saída de dados para a saída padrão (vídeo).
expressão : é qualquer combinação de caracteres ou variável.
```

Abaixo apresentamos alguns exemplos da utilização de cout. Note que em nenhuma das declarações da stream cout são formatados os dados a serem apresentados. Essa formatação é feita pela própria stream.

Variáveis	C++	Resultado
int x = 2;	Cout << "x = " << x;	x = 2
Float f = 1.2, g = 3;	Cout << f << " " << g;	1.20 3.00
Double dou = 2.14;	Cout << "valor = "	valor = 2.14
Char ch = 'F';	<< dou << "\nsex = " << ch;	sex = F

No último exemplo acima, mostramos o uso do caracter especial ‘\n’ para fazer com que o computador pule uma linha antes de continuar escrevendo dados na tela. Podemos utilizar todos os caracteres especiais definidos para printf() da mesma maneira que utilizamos agora ‘\n’. Uma lista completa destes caracteres está em 3.1.

O objeto cout utiliza flags de formatação para sinalizar as opções de formatação dos dados. Os flags (sinalizadores) de formato (definidos em <iomanip>) são os seguintes:

Manipulador	Significado
skipws	Ignora o espaço em branco na entrada
left	Saída ajustada à esquerda
right	Saída ajustada à direita
internal	Preenchimento após indicador de sinal ou base

dec	Conversão em decimal
oct	Conversão em octal
hex	Conversão em hexadecimal
showbase	Mostra o indicador de base na saída
showpoint	Mostra ponto decimal (para float)
uppercase	Saída hexadecimal maiúscula
showpos	Mostra o sinal '+' em inteiros positivos
scientific	usa notação científica de ponto flutuante 1.23E2
fixed	usa notação de ponto flutuante 1.23
unitbuf	Libera (flush) todas as streams depois da inserção
stdio	Libera (flush) stdout, stderr depois de inserção

A sintaxe para usar esses sinalizadores (ou flags) é a seguinte:

```
cout.setf(ios::<sinalizador>); // para ligar
cout.setf(ios::scientific);
cout << 12.345; // Imprimirá : 0.12345E2
cout.unsetf(ios::<sinalizador>); // para desligar
cout.unsetf(ios::scientific);
cout << 12.345 // Imprimirá: 12.345
```

O objeto cout permite estabelecer o tamanho de um campo para a impressão. Isto significa que podemos definir o número de colunas que serão ocupados por um valor ou texto a ser impresso. Geralmente, a definição de tamanho de campos é usada para alinhamento e estética de um relatório.

Os manipuladores de formato são utilizados para manipular a formatação dos dados em streams de saída. Alguns manipuladores são idênticos aos sinalizadores, a diferença está na forma mais compacta e na inclusão de outra biblioteca de classes <iomanip>.

Manipulador	Significado
dec	Passa para base decimal
oct	Passa para base octal
hex	Passa para base hexadecimal
ws	Extraí caracteres de espaço em branco
endl	Insere nova linha e libera stream
ends	Insere término nulo em string '\0'
flush	Libera o buffer de saída ostream alocado
setbase(n)	Ajusta o formato de conversão para a base n. default n = 0;
resetiosflags(long)	Limpa os bits de formato em <i>ins</i> ou <i>outs</i> especificadas.
setiosflags(long)	Ajusta os bits de formato em <i>ins</i> ou <i>outs</i> especificadas.
setfill(int n)	Ajusta o caractere de preenchimento para n
setprecision(int n)	Ajusta a precisão do ponto flutuante para n
setw(int n)	Ajusta o tamanho do campo para n

Exemplo do uso destes manipuladores:

```
// Exemplo do emprego de Cout
#include <iostream>
#include <iomanip>
using namespace std;

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

```
{
    float lap = 4.875;
    float bor = 234.5421234546;
    int can = 42;
    int cad = -8;
    cout << "\n\n" << setiosflags(ios::left);
    cout << setprecision(2);
    cout << "\n\t" << "Lapis" << setw(12) << lap;
    cout << "\n\t" << "Borracha" << setw(12) << bor;
    cout << "\n\t" << "Canetas" << setw(12) << can;
    cout << "\n\t" << "Cadernos" << setw(12) << cad;
    cout << "\n\t" << setfill('.') << "Fitas" << setw(12) << "TYE";

    return 0;
}
```

Observação: Todos esses cabeçalhos usados oferecem recursos no ambiente de nomes std, de modo que, para usar os nomes que eles oferecem, precisamos ou usar qualificação explícita com std: : ou trazer os nomes para o ambiente de nomes global com

using namespace std;

Observação: quando a função main() é chamada, esta recebe dois argumentos especificando o número de argumentos, usualmente chamado argc, e um array de argumentos, usualmente chamado de argv. Os argumentos são strings de caracteres, de modo que o tipo de argv é char*[argc+1]. O nome do programa é passado como argv[0], de modo que argc é sempre no mínimo 1. A lista de argumentos é terminada por um zero; isto é, argv[argc] = 0.

Observação: Se o tamanho do campo especificado em setw for menor que o tamanho mínimo necessário para imprimir o valor associado, a impressão utilizará o número necessário de colunas, ignorando o tamanho do campo.

Outro exemplo do emprego destes manipuladores:

```
// Exemplo do emprego de hex, dec e oct
#include <iostream>
int main(int argc, char* argv[])

{
    int n = 15;
    cout << "\n" << "Hexadecimal \t" << hex << n;
    cout << "\n" << "Decimal \t" << dec << n;
    cout << "\n" << "Octal \t" << oct << n;
    return 0;
}
```

13.4.2 A stream de entrada cin

A forma de entrar com dados em um programa C++ é através do objeto cin. Trata-se de um fluxo associado à entrada padrão do computador (stdin, i.e, o teclado). A sintaxe de cin é:

```
cin >> variável;
onde
```

>> : é o operador de extração usado para direcionar a entrada de dados à entrada padrão (teclado). Através do contexto, o compilador sabe que este operador não será o operador para deslocamento de bits mas sim o operador de leitura de dados.

variável : é o nome da variável onde desejamos guardar os valores lidos.

Na tabela a seguir, apresentamos alguns exemplos da leitura de dados através de `cin`. Note que em C++ não é necessário formatar o dado que está sendo lido. `cin` faz isto automaticamente, de acordo com a declaração da variável.

Variáveis	C++	C
<code>int x;</code>	<code>cin >> x;</code>	<code>scanf("%d",&x);</code>
<code>float f,g;</code>	<code>cin >> f >> g;</code>	<code>scanf("%f %f", &f, &g);</code>
<code>double dou; char ch;</code>	<code>cin >> dou >> ch;</code>	<code>scanf("%Lf %c", &dou, &ch);</code>

O objeto `cin` faz com que o programa aguarde que você digite o dado a ser adquirido e pressione a tecla [ENTER] para finalizar a entrada. O operador `>>` pode apresentar-se diversas vezes numa instrução com a finalidade de permitir a introdução de diversos valores ao mesmo tempo. Múltiplas entradas são digitadas separadas por um espaço em branco. O objeto `cin` entende um espaço em branco como término de uma entrada e o [ENTER] como finalizador geral.

Podemos ler números em outras bases numéricas, utilizando os manipuladores `hex`, `dec` e `oct` apresentados anteriormente.

```
int n = 0;
cin >> hex >> n;
```

A biblioteca `stream` define outras funções para leitura e escrita de dados que não serão aqui apresentadas. Todas as funções apresentadas na secção 3 podem ser aqui empregadas.

13.5 Funções

13.5.1 Valores Default Para Argumentos de uma Função

C++ permite a definição de valores default para argumentos de funções. Isto significa que, caso a chamada da função omita algum parâmetro, a função pode usar o default previamente definido. A forma de definir valores default é explicitá-los na declaração da função, omitindo ou não o nome das variáveis. Uma vez definidos na declaração, os valores default não devem ser repetidos na definição, pois o compilador reconhece nisto uma duplicidade de default. O exemplo a seguir apresenta a declaração do default de uma função e chamadas por ele viabilizadas:

```
unsigned int pares(int, int = 0);    // protótipo
int n1 = pares(20, 3);
int n2 = pares(20);
```

Se o primeiro argumento foi omitido, todos os subsequentes deverão ser omitidos. Se o segundo argumento for omitido, todos os subsequentes deverão ser omitidos e assim por diante. Podemos escrever funções que tenham parâmetros inicializados com um valor default e parâmetros não-inicializados, mas após a primeira inicialização todos os parâmetros seguintes devem ser inicializados. Exemplo:

```
void linha( int n = 20, char ch, int cor);    // Declaração inválida
void linha(int n, char ch = '*', int cor = 0); // Declaração válida.
```

13.5.2 Sobrecarga de Funções

Sobrecarregar funções significa criar uma família de funções com o mesmo nome, mas com a lista de parâmetros diferentes. Funções sobrecarregadas devem ter a lista de parâmetros diferentes ou em número ou em tipo. Quando a função é chamada, é a lista de parâmetros passada para ela que permite ao sistema identificar qual é o código adequado. Por exemplo, podemos definir a família de funções abaixo, lembrando que devemos definir cada função como se fosse única:

```
int cubo(int n);  
float cubo(float n);  
double cubo(double n);
```

13.5.3 Funções Inline

A palavra-chave `inline`, quando colocada como primeiro elemento do cabeçalho da definição de uma função, causa a inserção de uma nova cópia da função em todo lugar onde ela é chamada.

```
inline int cubo(int n);
```

A definição de uma função `inline` deve preceder a primeira chamada a ela. Ou seja, se a função for chamada em `main()`, seu código deve ser escrito antes de `main()`. Isto é necessário, pois o compilador deve conhecer de antemão o código da função para poder inseri-la dentro do programa.

Uma função é um código presente uma única vez no programa que pode ser executado muitas vezes. Assim, um dos motivos para escrever funções é o de poupar memória. Quando uma função é pequena e queremos aumentar a velocidade de execução de um programa, transformamo-la em `inline`.

13.5.4 Operador Unário de Referência: &

O operador de referência cria outro nome para uma variável já criada. As instruções:

```
int n;  
int &n1 = n; // toda referência deve ser inicializada
```

informam que `n1` é outro nome para `n`. Toda operação em qualquer dos nomes tem o mesmo resultado. Uma referência não é uma cópia da variável a quem se refere. É a mesma variável sob nomes diferentes.

O uso mais importante para referências é ao passar argumentos para funções. Os exemplos de argumentos de funções vistos até o momento são passados por valor ou por ponteiros.

Quando argumentos são passados por valor, a função chamada cria novas variáveis do mesmo tipo dos argumentos e copia nelas o valor dos argumentos passados. Desta forma, a função não tem acesso às variáveis originais da função que chamou, portanto não as pode modificar.

A principal vantagem da passagem por referência é a de que a função pode acessar as variáveis da função que chamou. Além desse benefício, este mecanismo possibilita que uma função retorne mais de um valor para a função que chama. Os valores a serem retornados são colocados em referências de variáveis da função chamadora.

A passagem por referência possui as vantagens da passagem de parâmetros por ponteiros, i.e., acesso direto à variável fornecida como parâmetro. No entanto, ponteiros exigem que tenhamos cuidado ao manipular o endereço das variáveis para não causar um erro fatal de acesso a memória. Este tipo de preocupação já não existe com passagem por referência, pois não lidamos com o endereço da variável mas sim com uma cópia do seu nome.

Funções que recebem argumentos por referência utilizam o operador `&` somente na definição do tipo do argumento e possuem chamadas idênticas a uma função normal. Exemplo:

```
void reajusta( float& num, float& p = 15 ); // Protótipo

int main(int argc, char* argv[])
{
    float preco = 10;
    reajusta( preco, 5);    // Chamada a função
    reajusta( preco);       // Chamada usando o argumento default
    return 0;
}

void reajusta( float& num, float& p ) //Definição
{
    num *= p;
}
```

13.6 Alocação Dinâmica de Memória em C++

A alocação dinâmica de memória em C++ é feita com os operadores `new` (alocação) e `delete` (liberação), análogos às funções `malloc()` e `free()`. Em C++, o gerenciamento dinâmico de memória é tão relevante que as palavras `new` e `delete` foram incorporadas a linguagem. A diferença básica entre os operadores C++ e C está na confiabilidade e facilidade de uso. Por exemplo, o tamanho do tipo para o qual se pede memória deve ser informado à `malloc()`, enquanto que `new` descobre esse tamanho automaticamente.

Declaração C:	Declaração C++
<code>double *ptr;</code>	<code>double *ptr;</code>
<code>pri = (double*)malloc(sizeof(double));</code>	<code>ptr = new double;</code>

C++ já fornece um ponteiro para o tipo de dado fornecido, não necessitando das conversões de tipo requeridas por `malloc()`. Outra vantagem é que C++ define quanto de memória é necessário, diminuindo o risco que o programador faça algum tipo de erro. Uma vantagem importante é que os operadores `new` e `delete` podem ser sobrecarregados (reimplementados) para um tipo de dado criado pelo programador. Isto permite que o programador controle como será alocada memória para o seu tipo de dado.

Sempre que uma memória alocada para uma variável não for mais necessária, devemos liberá-la utilizando o operador `delete`. As formas de utilizar o operador `new` (alocação de memória) e `delete` (liberação de memória) são:

- como operador ou função

```
ptr_tipo = new tipo;                ptr_tipo = new(tipo);
int * ptr1 = new int;                int * ptr2 = new(int);
delete ptr1;                          delete ptr2;
```
- alocação e liberação de matrizes

```
ptr_tipo = new tipo[tamanho_matriz];
int * ptr3 = new int[10];
delete[] ptr3;    // ou, explicitamente:    delete[10] ptr3;
```
- alocação e liberação de matrizes de ponteiros

```
ptr_ptr_tipo = new tipo * [tamanho_matriz];
int * * ptr4 = new int * [10];
delete[] ptr4;    // ou, explicitamente:    delete[10] ptr4;
```

Analogamente ao que ocorre a função `malloc()` do C, o operador `new` retorna um ponteiro nulo quando a memória disponível é insuficiente. Assim, devemos sempre verificar se a alocação foi realizada com sucesso, ou seja, se o ponteiro contém um endereço diferente de zero.

13.7 Exercícios

13.1 Escreva um programa que desenhe na tela o gráfico da função $y = \cos(x) + x/2$. Utilize `cout` e `cin` para isto. Desenhe todos os elementos do gráfico: título, legenda, linhas horizontal e vertical e numeração. Marque os pontos pertencentes ao gráfico com o caracter (*).

13.2 Crie um programa para fazer cadastro de doentes em um hospital. Utilize estruturas e funções, com passagem de parâmetros por referência. O programa deve apenas adquirir os dados e quando o usuário terminar o cadastro, o programa deve imprimir na tela todas as fichas feitas.

13.3 Modifique o programa acima para utilizar agora a alocação dinâmica em C++ e organize agora as fichas por ordem alfabética do nome do paciente.

14 Classes e Objetos em C++

No capítulo anterior começamos a apresentar a idéia da Orientação a Objetos em C++. Agora vamos começar a explicar o que são classes e objetos.

Suponha que estamos criando um programa de computador que jogue xadrez contra o usuário. Primeiramente, como no problema do arquiteto, temos um ambiente a modelar (no caso o jogo de xadrez) e vamos utilizar os componentes deste ambiente para descrevê-lo. Analisando-se rapidamente um jogo de xadrez, apontamos vários elementos que devem ser modelados.

Podemos modelar o tabuleiro definindo as suas dimensões, quantas posições livres existem, a cor do tabuleiro, definir métodos para desenhar este tabuleiro na tela e métodos para controlar as posições das peças no tabuleiro.

Podemos em seguida modelar todas as peças do jogo de xadrez. Cada uma destas peças será um objeto do nosso programa. Para cada peça, i.e. cada objeto peça, devemos definir algumas propriedades desta como tipo (peão, rei, rainha, torre, bispo e cavalo), de que time pertence (se pertence ao time do computador ou ao time do usuário), a cor desta peça e o seu desenho geométrico. Alguns métodos também são necessários como definir a maneira como esta peça pode ser movimentada no tabuleiro, a posição inicial desta peça no tabuleiro, como esta peça pode ser desenhada na tela, etc.

Aqui já fica bem clara a idéia do que sejam os objetos no jogo. O tabuleiro e cada uma de suas peças serão objetos no nosso programa. Desta forma, podemos modelar cada elemento isolado, permitindo assim uma boa estrutura para o programa e capacidade de reutilização.

Mas o que vem a ser as classes do nosso programa? Classes definem tipos de objetos. Quando modelamos as peças do tipo peão, percebemos que temos vários elementos no nosso ambiente com estas características, i.e., temos múltiplas instâncias do objeto peão. Então, definimos uma classe chamada peão e definimos nesta classe todos as propriedades e métodos para o elemento peão. Adicionamos um índice para podermos referenciar cada elemento peão isoladamente. Assim, definida a classe peão, podemos criar quantas instâncias (objetos) desta classe desejarmos. Todo objeto deve ser modelo a partir da definição da classe que o representa. Desta forma, temos que definir as classes para as demais peças do xadrez como o rei e a rainha.

Entretanto, podíamos ter criado somente uma classe intitulada: peças do xadrez. Desta forma não teremos uma classe para cada tipo de peça do xadrez mas uma classe que representa todos os tipos de peças de xadrez. Definir esta classe será uma atividade muito mais complexa e “insegura” (no sentido de poder gerar erros) que definir cada tipo de peça isoladamente. Por exemplo, nesta classe a dimensão da peça de xadrez e o método com que esta deve ser movimentada no tabuleiro vão ser diferentes para cada tipo de peça (peão, torre, bispo, ...). Isto nos leva a ter definições complexas para as atributos e métodos desta classe.

Devemos utilizar a primeira representação ou a segunda? Bom, depende do nível de abstração que desejamos (nível de detalhamento). Se o programador achar mais fácil tratar somente um tipo genérico de peça de xadrez, não há motivos para refinar este modelo. Se o programador desejar ter um controle diferenciado para cada tipo das peças, então é melhor refinar o modelo e criar uma classe para cada tipo. O bom senso é que define o quanto você deve ou não refinar um modelo, tome cuidado para não criar classes desnecessárias.

Você pode estar achando que terminamos, ou seja, que o jogo de xadrez já está modelado definindo-se o tabuleiro e as peças. Esta seria uma intuição normal, pois estes são os elementos que vemos quando jogamos xadrez. Mas não podemos esquecer de definir alguns elementos abstratos que estão inseridos no modelo. O primeiro é o próprio jogador. Nós queremos que o computador jogue contra o usuário, então devemos ter que modelar o jogador, definindo as suas propriedades como a cor da suas peças e os seus métodos, que aqui incluem as suas estratégias de jogo. Outro elemento necessário é uma espécie de juiz, que controlará as regras do jogo, saberá quando é a vez

do usuário ou do programador, isto é, devemos definir uma classe que controlará a execução do programa. Por fim, devemos ainda definir alguns elementos necessários para a interface do programa, como uma janela na tela onde desenharemos o tabuleiro, o mouse para que o usuário possa mexer as suas peças e todos os demais elementos necessários.

Para cada um destes elementos, deveremos definir as propriedades e métodos que segundo esta nossa aplicação podem modelar corretamente o nosso ambiente. Quando criamos as classes de objetos, também definimos as relações entre eles. Descrevendo os componentes do ambiente e a relação entre eles, já teremos então uma representação similar deste.

Neste capítulo seguimos apresentando a sintaxe para definição de classes de objetos em C++.

14.1 Tipo Classe e o Encapsulamento de Dados

A idéia fundamental de linguagens orientadas a objetos é a possibilidade de combinar num único registro campos que conterão dados e campos que são funções para operar os campos de dados do registro. Uma unidade assim é chamada classe.

Uma instância (variável) de uma classe é chamada objeto e conterá campos de dados e funções. Definir uma classe não cria nenhum objeto, do mesmo modo que a existência do tipo `int` não cria nenhuma variável inteira.

As funções de um objeto são chamadas funções-membro ou métodos e, de modo geral, são o único meio de acesso aos campos de dados também chamados de variáveis de instância.

Se o programa necessita atribuir um valor a alguma variável de instância, deve chamar uma função-membro que recebe o valor como argumento e faz a alteração. Devemos evitar o acesso a variáveis de instância diretamente.

Desta forma, os campos de dados estarão escondidos para nós, o que previne alterações acidentais ou inválidas. Dizemos então que os campos de dados e suas funções estão encapsulados (de cápsula) numa única entidade. As palavras encapsular e esconder são termos técnicos da definição de linguagens orientadas a objetos.

O conceito de “esconder” dados significa que eles estarão confinados dentro da classe e não poderão ser alterados, por descuido, por funções que não pertençam a classe.

Se alguma modificação ocorrer em variáveis de instância de um certo objeto, sabemos exatamente quais funções interagiram com elas: são as funções-membro do objeto. Nenhuma outra função pode acessar esses dados. Isso simplifica a escrita, manutenção e alteração do programa. Um programa em C++ consiste em um conjunto de objetos que se comunicam por meio de chamadas às funções-membro.

14.2 Definindo Classes

Vamos começar com um exemplo que mostra os elementos básicos na definição de classes e criação de objetos. O exemplo cria uma classe para representar um retângulo.

```
// Exemplo de uma classe
#include <iostream>
using namespace std;

class Retangulo // Define a classe
{
    private:
        int bas, alt;    // atributos privados
    public:
        static int n;    // atributos publicos

    // Construtores e Destrutores
```

```
Retangulo()                                { bas=0; alt=0; n++; }

Retangulo(int a, int b=0);

~Retangulo()                                { n--; }

// metodos da classe
void Init(int b, int h)    { bas = b; alt = h; }

void PrintData();

};

Retangulo::Retangulo(int a, int b)
{
    bas = a;
    alt = b;
    n++;
}

void Retangulo::PrintData() // Define funcao membro
{
    cout << "\nBase = " << bas << "  Altura = " << alt;
    cout << "\nArea = " << (bas*alt);
}

// inicializacao da variavel statica
int Retangulo::n = 0;

int main(int argc, char* argv[])
{
    Retangulo X(2,3);
    Retangulo Y[5]; // Declaracao de objetos
    Retangulo C[2] = { Retangulo(2,3), Retangulo(5,1) };

    Y[0].Init( 5, 3);           // Chama funcao membro de inicializacao

    X.PrintData();              // Chama funcao membro
    Y[1].PrintData();
    (C + 1)->PrintData();
    cout << "\n Quantidade de Objetos : " << C[1].n;
    return 0;
}
```

A classe Retângulo possui três atributos e 5 funções. Agrupar dados e funções numa mesma entidade é o fundamento da programação orientada a objetos.

A primeira tarefa na orientação a objetos é definir as classes. Uma definição de classe sempre começa pela palavra-chave `class` seguida do nome da classe (Retangulo, neste exemplo), de seu corpo delimitado por chaves e finalizado com um ponto-e-vírgula.

14.3 Membros Privados e Públicos

O corpo da definição da nossa classe contém as palavras `private` e `public` seguidas de dois-pontos. Elas especificam a visibilidade dos membros.

Os membros definidos após `private:` são chamados de parte privada da classe e podem ser acessados pela classe inteira, mas não fora dela. Um membro privado não pode ser acessado por meio de um objeto. Este conceito é equivalente à relação existente entre uma função e as suas variáveis locais (auto), elas não são visíveis fora da função. Geralmente, na parte privada da classe são colocados os membros que conterão dados. No entanto, pode-se colocar também funções que só poderão ser chamadas por outras funções da própria classe. Dizemos então que esses dados e funções estarão “escondidos”.

A seção pública da classe é formada pelos membros definidos após `public:` e pode ser acessada por qualquer função-membro e por qualquer outra função do programa onde um objeto foi declarado. Os membros públicos fazem a interface entre o programador e a classe. Geralmente, na parte pública de uma classe são colocadas as funções que irão operar os dados. Podemos, entretanto, colocar dados na parte pública da classe, entretanto estes dados poderão ser acessados diretamente pelos objetos desta classe. Então, não poderemos controlar o acesso a este dado da classe.

No exemplo acima, os dados `bas` e `alt` têm acesso privado, enquanto que o dado `n` e todas as funções têm acesso público.

14.4 Funções-Membro

Funções-membro, também chamadas métodos, são as que estão dentro da classe. Na classe `Retangulo`, definimos 5 funções-membro: `Retangulo()`, `Retangulo(int, int)`, `~Retangulo()`, `Init(int, int)` e `PrintData()`.

Estas funções executam operações comuns em classes: atribuem valores aos dados e imprimem certos resultados.

Observe que a função `Init(int, int)` recebe como parâmetro dois números inteiros e têm seu código definido dentro da definição da classe. Funções-membro de código definido dentro da classe são criadas como `inline` por default, tornando a execução desta mais rápida mas ocupando um espaço maior de memória para armazenar o programa.

Podemos, entretanto, definir funções-membro em algum lugar do programa fora da classe, contanto que o seu protótipo seja escrito dentro do corpo da definição da classe. Isto informa ao compilador que elas fazem parte da classe, mas foram definidas fora dela.

No nosso exemplo, definimos a função `PrintData()` fora da classe. Observe que na definição destas funções membro, o nome da função é precedido pelo nome da classe (no caso: `Retangulo`) e por um símbolo formado por dois caracteres de dois-pontos (`::`). Este símbolo é chamado operador de resolução de escopo. Este operador é o meio pelo qual informamos ao compilador que a função que está sendo definida está associada àquela classe particular. No nosso exemplo, o operador de resolução escopo informa que a função `PrintData()` é membro da classe `Retangulo`. Sendo assim, a definição completa fica:

```
void Retangulo::PrintData() // Define funcao membro
{
    cout << "\nBase = " << bas << "   Altura = " << alt;
    cout << "\nArea = " << (bas*alt);
}
```

14.5 Construtores & Destrutores

Muitas vezes é conveniente inicializar um objeto (instância da classe) quando ele é criado, sem necessidade de efetuar uma chamada à função-membro para que os dados sejam inicializados. A inicialização automática de um objeto é efetuada por meio da chamada a uma função-membro especial quando o objeto é criado. Esta função-membro é conhecida como construtor.

Um construtor é uma função-membro que tem o mesmo nome da classe e é executada automaticamente toda vez que um objeto é criado. Quando um objeto da classe **Retangulo** é criado, chama-se automaticamente o construtor da classe para inicializar os dados desta.

Observe que um construtor não deve retornar valor nenhum. O motivo é ele ser chamado diretamente pelo sistema e portanto não há como recuperar um valor de retorno. Por isso, é que declaramos um construtor, sem nenhum tipo, nas instruções :

```
Retangulo() { bas=0; alt=0; n++; }  
Retangulo(int a, int b=0);
```

No nosso exemplo, o construtor **Retangulo()** foi definido dentro da classe enquanto que o construtor **Retangulo(int, int)** foi definido fora desta, ficando claro na definição que um construtor não retorna nenhum tipo de dado.

Um construtor pode perfeitamente chamar uma função-membro. O seu código é igual ao de qualquer outra função, exceto pela falta de tipo.

Podemos sobrecarregar construtores para podermos criar objetos de forma diferenciada. No nosso exemplo fica clara esta aplicação. Podemos utilizar o construtor **Retangulo()** para inicializar os objetos com dados default, ou utilizar o construtor **Retangulo(int, int)** para explicitamente fornecer os valores iniciais para os dados da classe. Note que este segundo construtor faz uso de valores default para os parâmetros, permitindo assim que o usuário precise fornecer somente o primeiro parâmetro. O compilador decide qual construtor chamar de acordo com os parâmetros passados.

O construtor **Retangulo()** tem uma importância imensa em C++, é chamado de construtor default. Sempre que um objeto for criado, se nenhum outro construtor puder se utilizado (nenhum dado pode ser fornecido), então com certeza o construtor default será utilizado. Por exemplo, quando criamos uma matriz de objetos de uma classe. Não é interessante termos que inicializar cada objeto individualmente. Por isso, o programa utiliza o construtor default para criar todos os objetos juntos e inicializar todos com este construtor. Desta forma, podemos sempre garantir uma inicialização dos dados com o construtor default.

Outro elemento importante é o construtor de cópia, utilizado para iniciar um objeto como sendo uma cópia fiel de um objeto já existente. Este operador, pode ser definido passando-se como parâmetro do construtor o objeto que desejamos copiar. Aplicando no nosso exemplo, teríamos:

```
Retangulo(Retangulo Ret);
```

Uma função destrutora é chamada automaticamente toda vez que um objeto é destruído (liberado da memória) e tem o mesmo nome da classe precedido de um til (~):

```
~Retangulo() { n--; }
```

Da mesma forma que um construtor, os destrutores não podem ter valor de retorno. O destrutor também não pode receber argumentos (por isso, cada classe possui somente um construtor) nem pode ser chamado explicitamente pelo programador.

14.6 Criando Objetos

Após a definição da classe **Retangulo**, podemos declarar uma ou mais instâncias desse tipo. Uma instância de uma classe é chamada objeto, como já mencionamos anteriormente.

No nosso exemplo, criamos objetos da classe **Retangulo** através das instruções:

```
Retangulo X(2,3);  
Retangulo Y[5]; // Declaracao de objetos
```

```
Retangulo C[2] = { Retangulo(2,3), Retangulo(5,1) };
```

A primeira instrução cria um objeto da classe `Retangulo` chamado `X`, inicializando-o como o construtor `Retangulo(int, int)`.

A segunda instrução cria uma matriz unidimensional de objetos da classe `Retangulo`. Neste caso, o construtor default é empregado para a inicialização dos objetos.

A terceira instrução cria uma matriz unidimensional de objetos da classe `Retangulo`. A matriz é inicializada com um conjunto de objetos `Retangulo`. Note que a inicialização de matrizes de objetos mantém o formato anterior, ou seja, elementos delimitados por chaves, separados por vírgulas e o fim da instrução definido pelo (;) ponto-e-vírgula.

Observe que a definição da classe não cria nenhum objeto, somente o descreve. Este conceito é idêntico ao das estruturas que, quando definidas, não criam nenhuma variável. Declarar um objeto é similar a declarar uma variável de qualquer tipo; o espaço de memória é reservado.

14.7 Atributos do Tipo: Static

Cada objeto de uma classe possui a sua própria cópia dos dados da classe. Entretanto, muitas vezes desejamos que todos os objetos de uma classe tenham acesso um mesmo item de dado. Este item faria o mesmo papel de uma variável externa, porém será visível somente a todos os objetos daquela classe.

Quando um dado membro é declarado como `static`, é criado um único item para a classe como um todo, não importando o número de objetos declarados. Isto significa que o programa alocou somente um espaço de memória onde todos os objetos desta classe irão acessar este dado. A informação de um membro `static` é compartilhada por todos os objetos da mesma classe.

Um membro `static` é definido com um atributo qualquer da classe, tendo a mesma visibilidade de um membro normal da classe. Abaixo, mostramos a declaração de um membro `static` com visibilidade pública da classe `Retangulo`.

```
static int n;
```

Dados `static` devem ser obrigatoriamente redefinidos fora da definição da classe. Você pode considerar que a definição de um atributo da classe como `static`, simplesmente notifica o compilador que existe uma variável global (extern) que será utilizada por todos os objetos desta classe para armazenar este atributo. A declaração desta variável fora da classe é que irá realmente criar esta variável, alocando memória para esta. Esta declaração é feita em qualquer lugar fora da classe, da seguinte forma:

```
int Retangulo::n = 0;
```

Note que a palavra chave `static` não foi empregada e que utilizamos `Retangulo::` para indicar que `n` pertence à classe `Retangulo`, ou seja, só pode ser acessado através desta classe. Nesta declaração, podemos efetuar a inicialização desta variável, como apresentado acima.

Este tipo de variável pode ser bem útil. No programa exemplo, utilizamo-la para contar o número de objetos criados. Note que cada vez que criamos um objeto, chamamos o seu construtor que incrementará automaticamente `n`. Se destruímos um objeto, chamamos o seu destrutor que decrementará `n`. Desta forma, `n` conta automaticamente o número de objetos existentes de uma certa classe.

Observação: um dado do tipo `static` não pode ser inicializado no construtor de uma classe pois, se não, toda vez que criarmos um objeto desta classe, estaremos reinicializando o valor desta variável `static` para todos os demais objetos.

Funções podem ser definidas como `static`. Funções `static` têm características similares às variáveis `static`. São utilizadas para implementar recursos comuns a todos os objetos da classe.

Funções-membro static agem independentemente de qualquer objeto declarado. Conseqüentemente, estas funções não podem acessar nenhum membro não-static da classe. Funções membro static acessam somente membros static de classe. Funções static podem ser chamadas usando a mesma sintaxe utilizada para acessar dados static.

```
class moeda
{
    private:
        static float US;
    public:
        static void usvalor()
        {
            cout << "\nDigite o valor do dólar: ";
            cin >> US;
        }
};

moeda::usvalor();
```

14.8 Acessando Funções e Dados Públicos

Os membros públicos de um objeto são acessados por meio do operador ponto. A sintaxe é similar àquela de acesso aos membros de estrutura.

As funções-membro só podem ser chamadas quando associadas ao objeto específico pelo operador ponto. Uma função-membro age sobre um objeto particular e não sobre a classe como um todo. Nosso programa fornece alguns exemplos deste tipo de acesso:

Primeiro, acessando a função `Init(int, int)` do primeiro objeto da matriz `Y`:

```
Y[0].Init( 5, 3);           // Chama funcao membro de inicializacao
```

Aqui, acessando a função `PrintData()`:

```
X.PrintData();             // Chama funcao membro
Y[1].PrintData();
```

Note que podemos utilizar este operador para acessar dados públicos:

```
cout << "\n Quantidade de Objetos : " << C[1].n;
```

Como `'n'` é uma variável global (static) associada à classe `Retangulo`, podemos acessar este dado sem associá-lo a nenhum objeto ou até mesmo, sem nunca ter criado um objeto desta classe. Ao definir esta variável fora da classe, nos já alocamos memória para que ela fosse inicializada. Para acessá-la, basta utilizar o operador `(:)` para referenciar a classe associada a esta variável. Portanto, no caso de variáveis static, ambas instruções abaixo são equivalentes:

```
C[1].n           ==    Retangulo::n
```

O operador `(->)` pode ser utilizado quando temos um ponteiro para um objeto e queremos acessar um membro (dado ou função) público desta classe. No nosso exemplo, empregamos este operador na instrução abaixo:

```
(C + 1)->PrintData();
```

14.9 Objetos Const

O qualificador `const` na declaração de um objeto indica que o objeto é uma constante e nenhum de seus membros de dados podem ser alterados. Por exemplo, a instrução abaixo cria `X` como um objeto constante, ou seja, os dados de `X` não podem ser alterados em tempo de execução:

```
const Retangulo X(2,3);
```

Quando um objeto constante é declarado, o compilador proíbe a ele o acesso a qualquer função membro, pois não consegue identificar quais funções alteram os seus dados. Entretanto, há um modo de informar ao compilador que uma função-membro não altera nenhum dado do objeto e que poderemos chamá-la por meio de um objeto constante.

Ao colocar a palavra `const` após os parênteses que envolvem a lista de parâmetros da função, estaremos indicando que a função não modifica o objeto. A palavra `const` deve ser adicionada tanto no protótipo da função quanto na sua declaração. Desta forma é possível acessá-la a partir de um objeto `const`.

Por exemplo, vamos transformar a função `PrintData()` em `const`, para podermos utilizá-la junto com o objeto `X`:

```
class Retangulo
{
    void PrintData() const;
};
void Retangulo::PrintData() const // Define funcao membro
{
    cout << "\nBase = " << bas << "   Altura = " << alt;
    cout << "\nArea = " << (bas*alt);
}
```

14.10 Tipo Objeto

Objetos de classes podem ser usados normalmente como qualquer outro tipo de dado. Por isso podemos utilizá-los para definir matrizes de objetos ou para passar como parâmetro, da mesma forma que fazemos com outros tipos de dados.

As funções membro são criadas e colocadas na memória somente uma vez para a classe toda. As funções membro são compartilhadas por todos os objetos da classe. Já para os dados, a cada objeto que é declarado, uma nova região na memória é alocada de forma que cada objeto possui o seu conjunto de dados.

14.11 Exercícios

14.1 Crie um programa para calcular as médias das notas dos alunos de uma turma. Crie uma classe aluno e modele cada aluno com o seu conjunto de notas como sendo um objeto. Imprima na tela usando `cout`.

14.2 Usando a idéia da lista ligada apresentada capítulo de estruturas, crie um programa para representar restaurantes empregando uma lista ligada construída com classes.

14.3 Crie um programa que faça o controle de cadastro de usuários e controle de alocação para uma locadora de fitas de vídeo. Utilize uma classe para os clientes e outra para as fitas, e empregue o método da lista ligada para gerenciar um vetor dinâmico. O usuário deve poder incluir/retirar um cliente ou uma fita, fazer uma pesquisa, pedir a lista de clientes e a lista de fitas ou alterar algum parâmetro destas.

14.4 Escreva um programa para controlar o acesso das vagas de um estacionamento. Armazene dados importantes do carro como chapa do carro, a hora de entrada, a hora de saída. Controle a lotação do estacionamento e verifique se existem vagas disponíveis. O programa deve



gerar automaticamente o preço do estacionamento, quando o usuário retirar o carro. Não esqueça que alguns clientes podem ficar mais tempo do que o tempo notificado.

14.5 Modele o problema do jogo do xadrex apresentado no início do capítulo. Mas não jogue contra o computador mas contra um outro usuário usando o mesmo micro.

15 Sobrecarga de Operadores

O sentido do polimorfismo é o de um único nome para definir várias formas distintas. Em C++, chamamos de polimorfismo a criação de uma família de funções que compartilham do mesmo nome, mas cada uma tem código independente. O resultado da ação de cada uma das funções da família é o mesmo. A maneira de atingir esse resultado é distinta.

Como exemplo, suponhamos que desejamos calcular o salário líquido de todos os funcionários de uma empresa. Nessa empresa, há horistas, mensalistas, os que ganham por comissão, etc. Criamos um conjunto de funções em que cada uma tem um método diferente de calcular o salário. Quando a função é chamada, C++ saberá identificar qual é a função com o código adequado ao contexto.

A sobrecarga é um tipo particular de polimorfismo. Como exemplo, tomemos operador aritmético (+). Em C++, usamos esse operador para somar números inteiros ou para somar números reais:

```
5 + 4
3.7 + 12.5
```

O computador executa operações completamente diferentes para somar números inteiros e números reais. A utilização de um mesmo símbolo para a execução de operações distintas é chamada sobrecarga. O polimorfismo e a sobrecarga são habilidades únicas em linguagens orientadas a objetos.

Os operadores em C++ são definidos como funções normais onde o nome da função é o símbolo do operador e os parâmetros da função são os operandos do operador. Por exemplo, o operador de soma para números inteiros possui uma declaração do tipo:

```
int operator +(int a, int b);
```

Ou seja, a operação (a+b), é vista por C++ como uma chamada à função acima declarada. Quando escrevermos em nosso código esta operação, C++ utilizará a definição desta função para efetuar a operação. Note que a única diferença na declaração ou definição de um operador é que o nome de um operador é definido pela palavra chave **operator** seguida de espaço mais o símbolo que representa o operador, neste caso (+).

Assim, podemos tratar qualquer operador como sendo uma função normal mas que possui um nome especial e uma forma especial de chamada. Perceba, que quando escrevemos no código (a+b), C++ interpreta esta instrução como uma chamada a função **operator +** e passa para esta os parâmetros a e b.

Se os operadores são tratados por C++ como uma função qualquer, qual a razão da sua existência? Suponha, que tenhamos definido a seguinte função que executa a operação (a + b) :

```
int soma(int a, int b);
```

Qual das duas formas abaixo é mais clara e intuitiva para se escrever a soma de dois números:

```
int c = soma( a, b);
int c = a + b;
```

É claro que será segunda forma. Por isso, podemos tornar o nosso código fonte muito mais organizado e claro empregando-se os operadores ao invés de funções normais.

Assim como qualquer outra função em C++, os operadores podem ser sobrecarregados para outras aplicações, i.e., podemos redefinir os operadores para serem empregados em outros tipos de

dados criados pelo programador. Suponha que tenhamos criado uma classe para representar os números complexos chamada `complexo`. Então, com a sobrecarga do operador (+) poderemos redefinir a seguinte função para dois elementos da nossa classe:

```
complexo A, B;  
A = A + B;
```

Note, que esta definição é bem intuitiva e clara. Esta característica é muito importante para o programador, permitindo que este possa definir como os operadores existentes em C++ atuarão com relação aos tipos de dados definidos pelo programador.

Somente os operadores já existentes em C++ podem ser sobrecarregados (redefinidos) para operar sobre novos tipos de dados definidos pelo programador. Por exemplo, não podemos definir o símbolo (#) como um operador, pois este símbolo não é declarado por C++ como sendo um operador.

Sobrecarregar um operador significa redefinir o seu símbolo, de maneira que ele se aplique também a tipos de dados definidos pelo usuário como classes e estruturas. Sobrecarregar uma função significa utilizar um mesmo nome para tarefas parecidas, mas de códigos de programa diferentes. Várias funções com o mesmo nome podem apresentar diferentes implementações (trechos de programa). No instante em que você encontrar uma limitação pelo modo como os operadores C++ trabalham, pode mudá-los conforme as suas necessidades, por meio de sobrecargas.

15.1 A Sobrecarga como uma Função Global

A implementação de sobrecargas de operadores é definida por meio de funções chamadas operadoras. Estas funções podem ser criadas como membros de classes ou como funções globais, acessíveis a todo o programa.

A seguir vamos apresentar a sobrecarga do operador ++() incremento prefixado para a classe `Ponto`, criada para representar um ponto localizado no espaço bidimensional, por isso contém as coordenadas `x` e `y` do ponto.

```
//Ponto2 - Mostra a sobrecarga de operadores  
#include <iostream>  
using namespace std;  
  
class Ponto  
{  
public:  
    int X, Y;  
  
    Ponto(int aX = 0, int aY = 0)    { X = aX; Y = aY; }  
    void PrintPt() const {cout << '(' << X << ', ' << Y << ')';}  
};  
  
Ponto operator ++(Ponto ObjPonto) //incremento prefixado  
{  
    ++ObjPonto.X; ++ObjPonto.Y;  
    return ObjPonto;  
}  
  
int main(int argc, char* argv[])  
{  
    Ponto P1(1,2);  
    cout << "\n p1 = ";  
    P1.PrintPt();  
}
```

```
cout << "\n++p1 = ";  
(++P1).PrintPt();  
  
return 0;  
}
```

A classe **Ponto** é uma classe normal, possui dois atributos com acesso público, uma função construtora para inicializar os atributos e uma função para imprimir os dados na tela.

Sobrecarregamos o operador **++()** para incrementar os dados contidos em um objeto desta classe. Após a declaração da classe, segue a declaração do operador. Note que o operador é declarado como qualquer função global, a única diferença é a palavra-chave **operator** na frente do símbolo **++**. O operador recebe como parâmetro um objeto da classe **Ponto** e retorna também um objeto desta classe. O operador recebe somente um parâmetro, pois no caso, este é um operador unário e, por definição, só pode receber um único parâmetro. O operador acessa os dados públicos da classe **Ponto** e incrementa cada um deles, retornando o próprio objeto passado como parâmetro com o resultado da operação.

Em seguida, empregamos o novo operador incrementando o objeto **P1** e imprimindo na tela o resultado gerado por este operador.

15.2 Limitações e Características

A sobrecarga de operadores permite uma nova implementação de maneira como um operador funciona. Entretanto, é necessário respeitar a definição original do operador. Por exemplo, não é possível mudar um operador binário (que trabalha com dois operandos) para criar um operador unário (que trabalha com um único operando).

Não é permitido também estender a linguagem inventando nossos operadores representados com símbolos novos. Por exemplo, não se pode inventar uma operação usando o símbolo **##**. Este símbolo não é um operador válido em C++ e você não poderá torná-lo um operador. Devemos limitar-nos aos operadores já existentes.

A definição de operadores deve obedecer à precedência original do operador. Por exemplo, o operador de multiplicação (*****) tem precedência sobre o de adição (**+**). Não é possível modificar a precedência de operadores por meio de sobrecargas.

Nem todos os operadores podem ser sobrecarregados. Os seguintes operadores não podem ser sobrecarregados: o operador ponto de acesso a membros e o operador condicional ternário (**?:**).

Os operadores unários operam sobre um único operando. Exemplos de operadores unários são o de incremento (**++**), decremento (**--**) e menos unário (**-**). Por outro lado; os operadores binários operam sobre dois operandos (**+**, **-**, *****, **/**, **>**, **+=**, etc...). Por causa desta diferença, uma atenção especial deve ser dada ao uso de cada tipo.

Quando definimos um operador como sendo uma função-membro de uma classe, o primeiro parâmetro passado para a função operadora será considerado automaticamente como sendo um objeto desta classe. Declarar um operador como sendo membro de uma classe, permite que este tenha acesso a todos os dados desta classe, tenha as permissões de acesso como qualquer outra função membro e possa desfrutar das vantagens fornecidas pela herança de classes, apresentadas no próximo capítulo.

Na declaração de operadores como funções globais, os operadores unários receberão um único parâmetro enquanto que os operadores binários receberão dois parâmetros. Já na declaração de operadores como funções-membro de uma dada classe, já que o primeiro operando é sempre um objeto da classe, os operadores binários não receberão parâmetros enquanto que os operadores binários receberão um único parâmetro.

Uma função operadora deve ter definido um tipo de retorno, para que o resultado da operação com este tipo possa ser atribuídos a outras variáveis. Por exemplo, ao definir o operador $(a+b)$, temos que definir que esta operação retorna algum valor do tipo da variável c , para poder executar a seguinte instrução:

```
c = a + b;
```

15.3 Sobrecarga de Operadores como Função-Membro

No exemplo anterior, apresentamos a declaração do operador de incremento pré-fixado redefinido globalmente. Vamos agora, redefinir este operador com uma função-membro da classe Ponto.

```
#include <iostream>
using namespace std;

class Ponto
{
private:
    int X, Y;

public:
    Ponto(int aX = 0, int aY = 0)    { X = aX; Y = aY; }
    void PrintPt() const {cout << '(' << X << ', ' << Y << ')';}

    Ponto operator ++() //incremento prefixado
    {
        ++X; ++Y;
        Ponto Temp(X,Y);
        return Temp;
    }
};

int main(int argc, char* argv[])
{
    Ponto P1, P2(2,3);
    cout << "\n p1 = ";
    P1.PrintPt();
    cout << "\n++p1 = ";
    (++P1).PrintPt();

    cout << "\n++p2 = ";
    (++P2).PrintPt();
    P2 = ++P1;
    cout << "\n p2 = ";
    P2.PrintPt();

    Return 0;
}
```

Neste exemplo, efetuamos algumas modificações importantes. Os dados da classe Ponto, passaram a ter acesso privado, não podendo mais ser acessados por nenhuma função global.

Sobrecarregamos o operador incremento prefixado como sendo uma função-membro da classe Ponto, note que agora esta função não recebe nenhum parâmetro. A definição da função é

praticamente igual a anterior, retirando-se o parâmetro. Esta função é uma função inline da classe **Ponto**, por esta definida dentro da definição da classe. A função operadora `++()` agora cria um objeto temporário da classe **Ponto** para conter o resultado da operação a ser utilizado como retorno da função. Veremos a seguir, formas diferentes de retornar o resultado de uma função operadora.

Por fim, nada foi modificado no emprego do operador sobrecarregado. Note na função `main()`, que o emprego do operador não sofreu nenhuma modificação pelo fato deste ser declarado como uma função global ou uma função membro.

15.4 Estudo de Casos

Agora apresentaremos a sobrecarga de um grupo importante de operadores em C++ que servirão como exemplo para a sobrecarga de qualquer outro operador existente. Com este exemplo, discutiremos outros aspectos da sobrecarga e você perceberá também a utilidade desta ferramenta.

O primeiro exemplo cria uma classe para armazenar as notas dos alunos. A sobrecarga de dois operadores foi efetuada. C++ não faz a checagem de limites de matrizes. Por causa disto, a manipulação de matrizes em C++ apresenta várias falhas e permite erros acidentais de sobreposição de dados de memória se forem usados índices que ultrapassam o limite de dimensão da matriz. Por isso, efetuamos a sobrecarga do **operador []** para testar o limite da matriz. O outro operador, converterá um objeto da classe **Matriz** para um ponteiro float. Isto torna os objetos desta classe mais genéricos, podendo ser empregados facilmente para outras aplicações.

```
//Notas : Sobrecarga do operador []
#include <iostream>
#include <iomanip>
using namespace std;

const MAX = 50;

class Matriz
{
private:
    float n[MAX];

public:
    Matriz();

    float & operator [](int i); // sobrecarga []
    float Media(int i);
    operator float *() const // Funcao Conversora
    { return (float *)n; }
};

Matriz::Matriz()
{
    for(int i = 0; i<MAX; i++)
        n[i] = 0.0;
}

float & Matriz::operator [](int i)
{
    static float x = -1;
    if(i >= 0 && i < MAX)
        return n[i];
    else
```

```
{
    cout << "\nFora de Limite!";
    return x;
}
}
float Matriz::Media(int i)
{
    float m = 0.0;
    for(int j = 0; j < i; j++)
        m += n[j];
    return m/float(i);
}
int main(int argc, char* argv[])
{
    Matriz Notas;
    cout << setprecision(2);
    int i = 0;
    do
    {
        cout << "Digite a nota do aluno " << (i + 1) << " : ";
        cin >> Notas[i];
    }while (Notas[i++] >= 0);
    i--;
    float * Temp = Notas;
    cout << "\nNota do Segundo Aluno " << *(Temp + 1);
    cout << "\n\nMedia das notas : " << Notas.Media(i);
    return 0;
}
```

O programa cria uma matriz fixa para armazenar as notas dos alunos, imprimindo a média das notas no final. A novidade é a função `operator[]` que checa se o índice especificado está dentro do limite de dimensionamento da matriz. Se estiver, a função retorna uma referência ao elemento correspondente da matriz privada. Se não estiver, a função imprime uma mensagem de erro e retorna uma referência a uma variável `static float` com valor `-1`. Isto previne a sobreposição de outras regiões da memória caso o limite da matriz seja ultrapassado.

A função `operator[]` sobrecarrega o operador binário `[]` e está definida dentro da classe `Matriz`. Por isso, recebe um único parâmetro `int i`. A função é definida fora da classe, e por isso que escrevemos `Matriz::` antes do nome `operator[]` para indicar que se trata de uma função membro. Observe que a função `operator[]` retorna uma referência a um elemento da matriz `n`. Dessa forma, a expressão `notas[i]` age como o próprio elemento da matriz privada, podendo ser usado em instruções como :

```
cin >> notas[i];
notas[i] = 67.75;
```

Retornar uma referência não é simplesmente eficiente, mas sim necessário. Caso contrário, não teremos acesso direto a um elemento da classe mas a uma cópia deste, não podendo assim alterar o valor deste elemento com instruções de atribuição como as declaradas acima.

A função `operator[]` cria uma variável `static`, pois não podemos retornar uma referência para uma variável local (automática) já que esta será destruída ao final da execução da função.

Aqui começamos a discutir alguns aspectos sobre o operador de atribuição (`=`). Você sabe que o operador `=` pode atribuir o valor de uma variável simples a outra do mesmo tipo. Podemos também atribuir um objeto a outro da mesma classe usando instruções como:

```
Obj2 = Obj1;
```

Geralmente, quando o valor de um objeto é atribuído a outro de mesma classe, o valor de todos os seus membros de dados são simplesmente copiados no novo objeto. O compilador não executa nenhuma instrução especial no uso de atribuições entre tipos definidos pelo usuário.

A complexidade surge quando a atribuição é entre variáveis de tipos diferentes. Podemos converter um valor float para uma variável inteira através de uma das seguintes instruções:

```
int i = int(3.141567);  
int i = (int) 3.141567;
```

Assim como podemos converter um tipo básico em outro tipo, podemos também converter um objeto de uma classe em um tipo básico ou em um objeto de outra classe.

Para converter um tipo definido pelo programador, como classes, num tipo básico é necessário sobrecarregar o operador de molde, criando uma função chamada conversora. No exemplo anterior, sobrecarregamos o operador float `*`() como sendo uma função da classe `Matriz`, para converter um objeto desta classe em um ponteiro para uma lista de valores float:

```
operator float *() const // Funcao Conversora  
{ return (float *)n; }
```

Este operador foi chamado pela instrução:

```
float * Temp = Notas;
```

que poderia ter sido escrita na forma explícita:

```
float * Temp = float *(Notas);
```

As duas formas têm exatamente o mesmo efeito. Primeiro o compilador procura pela sobrecarga da operação de atribuição `=()` para este tipo, se não encontra, então o compilador procura pela função conversora para este tipo. Note que a função conversora não declara um tipo de retorno, o próprio nome da função já define o tipo de retorno.

No próximo exemplo vamos adicionar um conjunto de operadores a classe `Ponto` definida anteriormente, para apresentar outras características e exemplos da sobrecarga de operadores.

```
// Classe Ponto ... exemplo da sobrecarga de operadores  
#include <iostream>  
using namespace std;  
enum bool { false, true };  
  
class Ponto  
{  
public:  
    int X, Y;  
  
    Ponto(int aX = 0, int aY = 0)    { X = aX; Y = aY; }  
  
    Ponto operator ++(); // prefixado  
    Ponto operator ++(int); // pos-fixado  
    Ponto operator +(Ponto const aP) const;  
    Ponto operator +=(int const aN);  
    bool operator ==(Ponto const aP) const;  
    Ponto operator =(int aN);  
};  
  
Ponto Ponto::operator ++() //prefixado
```

```
{
    ++X; ++Y;
    return *this;
}
Ponto Ponto::operator ++(int) //pos-fixado
{
    ++X; ++Y;
    return Ponto(X-1, Y-1);
}
Ponto Ponto::operator +(Ponto const aP) const
{
    return Ponto(X + aP.X, Y + aP.Y);
}
Ponto Ponto::operator +=(int const aN)
{
    X += aN; Y += aN;
    return *this;
}
bool Ponto::operator ==(Ponto const aP) const
{
    return ((X == aP.X)&&(Y == aP.Y));
}
Ponto Ponto::operator =(int aN)
{
    X = aN;
    return *this;
}

class PFloat
{
public:
    float X, Y;

    PFloat(float aX = 0, float aY = 0)
    { X = aX; Y = aY; }
    operator Ponto() const // Converte tipo
    {
        return Ponto((int) X, (int) Y);
    }
};

// Operadores para objetos Iostream
ostream & operator<<(ostream & OS, Ponto const aP)
{
    OS << '(' << aP.X << ', ' << aP.Y << ')';
}

int main(int argc, char* argv[])
{
    Ponto P1, P2(2,3), P3;
    P3 = 2;
    PFloat Pf( 2.12, 3.14);
    P1 = Pf;
    cout << "\n p1 = " << P1;
```

```
cout << "\n ++p1 = " << ++P1;  
cout << "\n p2 == p1 ? -> " << (P1 == P2);  
cout << "\n p2 = " << P2;  
cout << "\n p2 + p1 = " << (P2+P1);  
cout << "\n pf = " << Pf;  
cout << "\n p3 = " << P3;  
  
return 0;  
}
```

Começamos definindo um tipo `enum`, para representar os valores booleanos.

A classe `Ponto` possui agora os atributos `X` e `Y` como sendo dados públicos, somente para facilitar o acesso a estas variáveis. `Ponto` possui também um construtor default para a inicialização dos dados.

Os primeiros operados sobrecarregados no exemplo são os operadores unários de incremento pré-fixado e pós-fixado. Dentre os operadores unários existentes em C++, a sobrecarga dos operadores de incremento e decremento apresenta uma dificuldade maior pelo fato de eles operarem de forma diferente quando prefixados ou pós-fixados.

A declaração :

```
Ponto operator ++(); // prefixado
```

notifica o compilador da sobrecarga do operador prefixado. A definição do operador é idêntica à definição apresentada anteriormente. A primeira diferença é que agora o operador está definido fora da definição da classe. A segunda diferença está na forma como retornamos o resultado da operação. Podemos fazê-la de três formas:

- a) `Ponto Temp( X, Y);`
`return Temp;`
- b) `return Temp( X, Y);`
- c) `return *this;`

As duas primeiras formas criam um objeto temporário, armazenam nele os valores obtidos na operação e retornam este objeto temporário. Neste caso, não podemos utilizar a passagem de parâmetros por referência pois estes objetos temporários serão destruídos ao término da função.

O terceiro utiliza o ponteiro especial **this**. O ponteiro `this` é um ponteiro especial que pode ser acessado por todas as funções-membro do objeto. Este ponteiro aponta para o próprio objeto. Qualquer função-membro pode acessar o endereço do objeto do qual é membro por meio do ponteiro `this`. Quando um objeto é criado, o compilador atribui o endereço do objeto ao ponteiro `this`. Usamos o ponteiro `this` quando queremos que uma função-membro retorne o próprio objeto do qual é membro. Para retornar o próprio objeto, devemos utilizar `*this`.

Após o operador de incremento pré-fixado, definimos o operador de incremento pós-fixado. Mas como pode o compilador distinguir os dois, sendo que ambos utilizam o mesmo simbolo e possuem o mesmo operando? Neste caso, C++ adicionou um parâmetro inteiro ao operador de incremento pós-fixado, para distingui-lo do pré-fixado. Este parâmetro `int` não serve para nada, a não ser para podermos distinguir a declaração dos dois tipos de operadores. Por isso, é que declaramos este operador da seguinte forma:

```
Ponto operator ++(int); // pos-fixado
```

Observe que esta função operadora é definida como a anterior. Entretanto, agora não podemos utilizar o ponteiro `this` para retornar o próprio objeto mas temos que criar um objeto

temporário que conterá o valor a ser retornado. Isto ocorre por que no operador pós-fixado, primeiro nós utilizamos a variável para depois incrementá-la.

Em seguida, temos a definição dos operadores binários aritméticos (+) e (+=). Um operador binário definido como uma função-membro possuirá somente um parâmetro, como foi dito anteriormente. Uma classe pode redefinir várias vezes os operadores binários, desde que em cada definição tenha um tipo de dado diferente como operando. No exemplo, o operador (+) implementa a soma de dois objetos da classe `Ponto`, enquanto que o operador (+=) implementa a soma de um valor inteiro a um objeto da classe `Ponto`, armazenando o resultado no próprio objeto. O operador (+) cria um objeto temporário para conter o resultado enquanto que o operador (+=) utiliza o ponteiro `this` para retornar o próprio objeto como resultado.

Depois, temos a sobrecarga do operador binário lógico (==) . Como os operadores acima, este recebe um único parâmetro, que no caso será outro objeto da classe `Ponto`. O tipo de retorno aqui é um tipo booleano (inteiro), já que o operador é do tipo lógico.

O programador tem liberdade em C++ para definir a aplicação que será dada para um operador. Por exemplo, um programador pode sobrecarregar um operador binário de bit (^=) para atuar como um operador lógico entre objetos de uma dada classe, retornando um valor do tipo inteiro. Não significa que um operador lógico, somente poderá ser sobrecarregado para efetuar operações lógicas entre objetos. Podemos, por exemplo, redefinir um operador lógico para efetuar uma operação aritmética ou qualquer outra operação entre dois objetos. C++ não controla a maneira como utilizamos os operadores com os novos tipos de dados. C++ somente não permite a criação de novos operadores e solicita que o número de parâmetros definido para cada operador seja respeitado. O que o operador faz, é de responsabilidade puramente do programador.

Seguindo com o nosso exemplo, o próximo operador definido é um operador de atribuição (=), utilizado para atribuir um inteiro a um objeto do tipo `Ponto`, possuindo a seguinte declaração:

```
Ponto operator =(int aN);
```

O operador atribui a dimensão X do objeto ponto o valor de aN. Note que a forma de declarar e definir o operador é a mesma utilizada nos exemplos acima. Este operador também utiliza o ponteiro `this` para retornar o próprio objeto. A novidade aqui aparece quando utilizamos este operador para converter um inteiro em um objeto do tipo `Ponto`. Esta é uma das formas importantes para converter um tipo em outro, e aparece quando executamos a seguinte instrução e atribuímos 2 ao objeto `Ponto P3`:

```
P3 = 2;
```

Em seguida, criamos a classe `PFloat` parecida com a classe `Ponto`, para exemplificar a conversão de objetos de uma classe em objetos de outra classe. Note que `PFloat` contém as coordenadas de um ponto x e y com valores float. Esta classe, possui também um construtor default. A novidade surge na declaração do operador de conversão para objeto da classe `Ponto`:

```
operator Ponto() const // Converte tipo
{
    return Ponto((int) X, (int) Y);
}
```

No exemplo anterior apresentamos a conversão de um objeto da classe `Matriz` para um ponteiro do tipo `float`. Aqui, mostramos como podemos criar uma função conversora para converter um objeto de uma classe em outra classe. A função `operator Ponto() const` converterá um objeto da classe `PFloat` para um objeto da classe `Ponto`. A mesma sintaxe pode ser aplicada para converter qualquer tipo de objeto em uma outra classe.

Poderíamos também implementar esta conversão de outras duas formas diferentes.

- 1) Sobrecarregando o operador de atribuição da classe `Ponto`, para atribuir um objeto da classe `PFloat` em um objeto da classe `Ponto`.
- 2) Criando um construtor na classe `Ponto`, que receba como único parâmetro um objeto da classe `PFloat`:

```
Ponto::Ponto(PFloat P1) { X =(int)P.X; Y =(int)P.Y;}
```

O resultado desta conversão aparece na execução da seguinte instrução:

```
P1 = Pf;
```

Por fim, sobrecarregamos o operador global de inserção de dados em objetos da classe `ostream`. Como vimos nos capítulos anteriores, as classes `stream` controlam o acesso a entradas e saídas do computador. Em especial, a classe `istream` controla a entrada padrão e a classe `ostream` controla a saída padrão. Lembrando, que o objeto `cout` pertence à classe `ostream` e o objeto `cin` a classe `istream`. As classes `stream` redefiniram os operadores `<<` e `>>` para operar sobre todos os tipos básicos existentes em C++. Desta forma, podemos escrever instruções como:

```
cout << "Bom Dia!"  
cin >> n;
```

chamando automaticamente as funções de leitura e escrita da classe `stream` através da sobrecarga de operadores. Na versão anterior da classe `Ponto`, utilizávamos a função `PrintPt()` para imprimir o valor de um objeto desta classe na tela. Entretanto, podemos fazer isto de uma forma muito mais elegante definindo o operador `<<` para receber como operandos um objeto da classe `ostream` e um objeto da classe `Ponto`.

```
// Operadores para objetos Iostream  
ostream & operator<<(ostream & OS, Ponto const aP)  
{  
    OS << '(' << aP.X << ', ' << aP.Y << ')';  
}
```

Na sobrecarga deste operador, definimos como o objeto da classe `Ponto` será apresentado na tela. Definida esta sobrecarga, podemos então passar um objeto da classe `Ponto` para um objeto `cout`, como fazemos com qualquer tipo básico. O mesmo princípio pode ser utilizado para a sobrecarga do operador `>>` associando um objeto da classe `istream` (`cin`) a um objeto da classe `Ponto`.

Podemos verificar o emprego desta função sobrecarregada através da instrução abaixo, que imprime na tela uma string de caracteres e, em seguida, os dados de um objeto `Ponto`:

```
cout << "\n p1 = " << P1;
```

Entretanto, uma dúvida surge, o que acontece na linha de código abaixo:

```
cout << "\n pf = " << Pf;
```

Onde `Pf` é um objeto da classe `PFloat` mas não sobrecarregamos o operador `<<` para receber como segundo parâmetro um objeto da classe `PFloat`. Neste caso, o programa utiliza a função de conversão de objeto `PFloat` em objetos do tipo `Ponto` definida acima, para converter este objeto e utilizar a função operadora `<<` definida para objetos do tipo `Ponto`. Este exemplo, mostra a importância e flexibilidade das conversões de tipos.

15.5 Exercícios

15.1 Acrescente a classe Ponto à sobrecarga dos operadores: menos unário (-), menos binário (-), multiplicação (*), divisão (/), módulo (%), às operações aritméticas de atribuição e às operações lógicas.

15.2 Crie uma classe que defina um número complexo ($x + yi$, onde x e y são do tipo float e i é a raiz quadrada de -1). Defina, todos os os operadores aritméticos, de bits e lógicos para esta classe. Sobrecarregue os operadores $<<$ e $>>$ da classe stream, para a leitura/escrita de objetos desta classe. Crie uma outra classe para a representação de números complexos no plano polar. Crie funções de conversão, para que os objetos das duas classes sejam equivalentes.

15.3 Crie uma classe para representar strings. Reimplemente os operadores aritméticos, de atribuição e lógicos para o contexto desta classe. Crie funções conversoras dos tipos básicos (int, char, float, ...) para objetos desta classe. Sobrecarregue o operador $[]$ para controlar o acesso a elementos individuais da classe string e crie uma função que converta objetos da classe string em um ponteiro do tipo char.

15.4 Inclua uma sobrecarga do operador de chamada à função () na classe string anterior. O seu protótipo é o seguinte:

```
void operator()(int n, char ch);
```

Esta sobrecarga atribui o caracter ch ao n-ésimo caracter da cadeia. Escreva um programa de teste.

15.5 Crie uma classe para representar as cores (class Cor). Sobrecarregue os operadores aritméticos e relacionais para esta classe. Utilize o princípio, que a soma de duas cores é uma cor misturada com metade de cada uma das cores anteriores. Defina as outras funções aritméticas. Utilize os operadores para os objetos cout e cin, para ler e escrever dados desta classe.

16 Herança

Nos capítulos anteriores definimos que classe representa um tipo de objeto. Então, quando modelamos um ambiente através da orientação a objetos, criamos uma classe para cada tipo de objeto presente e modelado no ambiente. Entretanto, existem alguns tipos (classes) de objetos que possuem características semelhantes com outros tipos (classes) de objetos.

Por exemplo, quando estamos modelando o funcionamento de um banco, podemos definir como objetos do modelo o cliente que chega na agência e os funcionários do banco. Ao modelarmos estas duas entidades, percebemos que ambas possuem determinadas características em comum; ambas entidades são seres humanos. Por serem pessoas, ambas as entidades conterão necessariamente a descrição de uma pessoa mais os seus dados específicos. Para não representar duas vezes o elemento pessoa, uma na entidade cliente e outra na entidade funcionário, criamos então uma entidade separada chamada pessoa e descrevemos através desta a entidade pessoa. Após a definição desta nova entidade, dizemos que as classes funcionários e clientes herdam da classe-base pessoa todas as características que definem um ser humano no nosso modelo. Na figura abaixo apresentamos a diferença entre as duas modelagens.

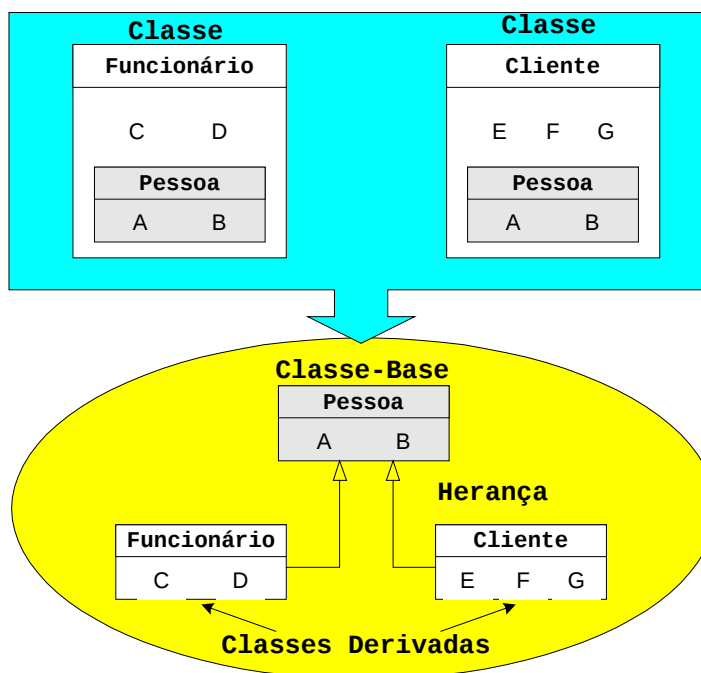


Figura 3 : Modelagem aplicando-se o princípio da Herança.

O processo de hierarquia está presente quando dividimos classes em sub-classes, mantendo-se o princípio de que cada subclasse herda as características da classe da qual foi derivada. Além das características herdadas, cada subclasse tem suas características particulares. Em programação orientada a objetos, o conceito de subclasse ou classes derivadas é chamado de herança.

Em C++, a classe de origem é chamada classe-base e as classes que compartilham as características de uma classe-base e têm outras características adicionais são chamadas de classes derivadas (ver Figura 3). Em C++, definimos uma classe-base quando identificamos características comuns em um grupo de classes.

Com o princípio da herança, além de precisarmos modelar somente uma vez a entidade pessoa, o nosso modelo ainda ganha em flexibilidade e em organização do projeto. Suponha que

neste modelo você tenha que futuramente incluir a entidade “acionista” no modelo do banco. Bem, considerando que todo acionista também é um ser humano, basta definir que a classe acionista será derivada da classe-base pessoa, herdando todas as suas propriedades (métodos e atributos), e adicionar a classe acionista as suas propriedades particulares.

Podemos ainda refinar o modelo acima, criando diferentes classes para modelar diferentes tipos de clientes. Podemos criar uma classe para modelar os clientes que são universitários. Esta classe seria um refinamento da classe cliente e, por isso, representamos esta entidade como uma classe derivada da classe cliente. Neste sentido, somente precisamos adicionar as propriedades particulares da classe cliente-universitário como: o número de matrícula, o nome do curso e a data prevista para a formatura. Uma classe que é derivada de uma classe-base pode, por sua vez, ser a classe-base de outra classe.

Outro aspecto importante é a reutilização do programa. Suponha que amanhã apareça um projeto para modelarmos uma célula de produção de uma fábrica. Neste modelo, precisamos modelar pessoas com diferentes propriedades: engenheiros, gerentes e funcionários. Neste caso, podemos utilizar a nossa classe-base pessoa para definir estas três novas classes, ganhando assim em tempo de projeto. Se recebermos um outro novo projeto para representar uma padaria, poderemos então aproveitar diversas entidades do nosso modelo como as classes pessoa, cliente, funcionário e cliente-universitário. Diminuindo, assim, consideravelmente o tempo deste projeto.

Você pode então criar uma biblioteca com todas as classes definidas que poderão vir a ser o núcleo de muitos programas. Desta forma, você terá grandes vantagens em termos de reutilização e baixo custo de projeto. O uso de uma biblioteca de classes oferece uma grande vantagem sobre o uso de uma biblioteca de funções: o programador pode criar classes derivadas de classes-base da biblioteca. Isto significa que, sem alterar a classe-base, é possível adicionar a ela características diferentes que a tornarão capaz de representar exatamente o que desejamos. A facilidade com que classes existentes podem ser reutilizadas sem ser alteradas é um dos maiores benefícios oferecidos por linguagens orientadas a objetos.

O processo de herança vai além da derivação simples. Uma classe derivada pode herdar características de mais de uma classe-base. Classes derivadas não estão limitadas a herança única.

Sempre que estivermos modelando um ambiente, seja este real ou abstrato, precisamos fazê-lo da forma mais genérica possível. Quando mais genérico for o modelo, mais flexível será o nosso sistema e maior será a reusabilidade e expandibilidade do nosso sistema. Modelar um sistema significa representar todas as propriedades importantes deste e não descrever este da maneira mais detalhada possível. Não exagere no nível de detalhe do seu modelo. Isto diminuirá a flexibilidade do seu modelo, aumentará o tempo de projeto, aumentará a quantidade de erros, diminuirá a velocidade do programa e aumentará o tamanho deste.

A orientação a objetos é realmente muito poderosa, permitindo-nos a construção de projetos flexíveis, permitindo a reutilização de partes do modelo e do código bem como a fácil expansão do sistema. Entretanto, a eficiência do seu programa orientado a objetos depende diretamente da qualidade do seu modelo. Modelos enxutos e bem definidos, geram programas rápidos e flexíveis. Modelos enormes e obscuros, geram programas “gordos” e complexos, cheios de erros, ilegíveis e que não podem ser reaproveitados. Tenha consciência ao modelar um sistema e bom-senso para definir as classes e as suas relações. O bom entendimento da metodologia orientada a objetos, permitirá que você modele adequadamente um sistema qualquer.

16.1 Derivando uma Classe

Agora mostraremos como podemos implementar em C++ a criação de uma classe-base e a sua derivação criando uma nova classe. Aqui, apresentaremos na forma de um exemplo a sintaxe empregada em C++. Como exemplo, vamos modelar a classe `PESSOA` apresentada como exemplo acima e a classe cliente, derivada da classe `PESSOA`.

```
// Exemplo de classes derivadas ...
#include <iostream>
#include <string>
using namespace std;
#define Nome_Size 80

class Pessoa
{
protected:
    char *      Nome;
    int         Idade;
    long int    RG;

public:
    Pessoa();
    Pessoa(char * aNome, int aId = 0, long int aRG = 0);
    Pessoa(Pessoa const & aP);
    ~Pessoa();

    char const * GetNome()      const { return Nome; }
    int          GetIdade()     const { return Idade; }
    long int     GetRG()        const { return RG; }

    void         GetData();
};

Pessoa::Pessoa()
{
    Nome = new char[Nome_Size];
    Idade = 0;
    RG = 0;
}

Pessoa::Pessoa( char * aNome, int aId, long int aRG)
{
    Nome = new char[Nome_Size];
    strcpy( Nome, aNome);
    Idade = aId;
    RG = aRG;
}

Pessoa::Pessoa(Pessoa const & aP)
{
    Nome = new char[Nome_Size];
    strcpy( Nome, aP.GetNome());
    Idade = aP.GetIdade();
    RG = aP.GetRG();
}

Pessoa::~~Pessoa()
{
    if(Nome != 0)
    {
        delete Nome;
        Nome = 0;
    }
}

void Pessoa::GetData()
{

```

```
        cout << "\nNome : ";
        cin.getline( (char *)Nome, Nome_Size);
        cout << "Idade : ";
        cin >> Idade;
        cout << "RG : ";
        cin >> RG;
    }

ostream & operator <<(ostream & OS, Pessoa & const P)
{
    OS << "\n\n Dados Pessoais -----"
    << "\n Nome   : " << P.GetNome()
    << "\n Idade  : " << P.GetIdade()
    << "\n RG    : " << P.GetRG();
    return OS;
}

class Cliente : public Pessoa
{
protected:
    int  Conta;
    int  Saldo;
public:
    Cliente();
    Cliente(char * aNome, int aConta = 0, int aId = 0, long int aRG =
0, int aSaldo = 0);

    int GetConta() const { return Conta; }
    int GetSaldo()  const { return Saldo; }

    void          GetData();
};

Cliente::Cliente() : Pessoa()
{
    Conta = 0;
    Saldo = 0;
}

Cliente::Cliente(char * aNome, int aConta, int aId, long int aRG, int
aSaldo) : Pessoa(aNome, aId, aRG)
{
    Conta = aConta;
    Saldo = aSaldo;
}

void Cliente::GetData()
{
    Pessoa::GetData();
    cout << "Conta : ";
    cin >> Conta;
    cout << "Saldo : ";
    cin >> Saldo;
}

ostream & operator <<(ostream & OS, Cliente & const C)
{
    OS << Pessoa(C);
    OS << "\n Conta : " << C.GetConta()
}
```

```
<< "\n Saldo : " << C.GetSaldo();
return OS;
}
int main(int argc, char* argv[])
{
    Pessoa P1("Carlos", 18, 1315678);
    Cliente C1, C2("Pedro", 1234, 17, 123432);
    C2.GetData();
    C1.Pessoa::GetData();
    Pessoa P2 = P1;
    Pessoa P3 = C1;
    cout << P1 << P2 << P3;
    cout << C1 << C2;
    return 0;
}
```

Para derivar uma nova classe de uma classe já existente, basta indicar o nome da classe-base após o nome da nova classe, separado por dois-pontos (:). O nome da classe-base pode ser precedido da palavra `public` ou da palavra `private`.

```
class Cliente : public Pessoa
```

Esta declaração indica que a classe `Cliente` é derivada da classe `Pessoa`. A classe `Cliente` incorpora todos os membros da classe `Pessoa`, além de seus próprios membros.

Nem sempre queremos criar objetos da classe-base. Muitas vezes declaramos classes somente para que estas sirvam como uma classe-base para um conjunto de classes derivadas. Classes usadas somente para derivar outras classes são geralmente chamadas classes abstratas, o que indica que nenhuma instância (objeto) delas é criado. Entretanto, o termo abstrato tem uma definição mais precisa quando usado com funções virtuais, apresentado no próximo capítulo.

16.2 Dados Protected

Observe que a parte pública da classe-base está disponível à classe derivada e a qualquer objeto dessa classe. Qualquer membro público da classe-base é automaticamente um membro da classe derivada. Entretanto, nenhum membro privado da classe-base pode ser acessado pela classe derivada. Por isso, criaram-se os membros **protected**. Membros `protected` são semelhantes a membros `private`, exceto pelo fato de que são visíveis a todos os membros de uma classe derivada. Observe que membros `protected` não podem ser acessados por nenhum objeto declarado fora dessas classes. Sempre que você escrever uma classe que pode vir a ser uma classe-base de outras classes, declare como `protected` os membros privados necessários às classes derivadas.

Segundo este princípio, os dados definidos na classe-base `Pessoa` estarão disponíveis a funções-membro da classe `Cliente`, porque foram definidas como `protected`. Entretanto, terão restrições do tipo `private` para acessos fora da classe-base ou fora da classe derivada.

16.3 Construtores & Destrutores

Se nenhum construtor for especificado na classe derivada, o construtor da classe-base será usado automaticamente. Observe os construtores da classe `Cliente`:

```
Cliente::Cliente() : Pessoa()
{
    Conta = 0;
    Saldo = 0;
}
```

```
Cliente::Cliente(char * aNome, int aConta, int aId, long int aRG,
    int aSaldo) : Pessoa(aNome, aId, aRG)
{
    Conta = aConta;
    Saldo = aSaldo;
}
```

Estes construtores têm uma sintaxe não familiar: os dois-pontos seguidos do nome da função construtora da classe-base. Isto causa a chamada ao construtor sem argumentos da classe **Pessoa**, quando um objeto é criado sem valores de inicialização, e a chamada ao construtor com argumentos da classe **Cliente**, quando fornecemos valores para a inicialização do objeto. A instrução:

```
Cliente C2("Pedro", 1234, 17, 123432);
```

usa o construtor com argumentos da classe **Cliente**. Este construtor chama o construtor correspondente da classe-base, enviando os argumentos recebidos para o construtor da classe-base **Pessoa** que, por sua vez, inicializa o objeto. Note, que o construtor da classe derivada chama o construtor da classe-base e depois executa algumas rotinas para inicializar somente os dados específicos da classe-derivada.

No nosso programa exemplo, a classe **Pessoa** tem um atributo do tipo ponteiro que é alocado dinamicamente quando criamos um objeto e é destruído (liberado) quando destruímos este objeto. Para tal, incluímos a seguinte instrução nos construtores da classe:

```
Nome = new char[Nome_Size];
```

Então, definimos um destrutor para a classe para liberar a memória alocada antes de destruir o objeto. Observe, que antes de liberarmos a memória vamos testar se o ponteiro atual é válido. Mas a classe **Cliente** não define nenhum destrutor. Isto implica que quando destruímos um objeto da classe **Cliente**, o compilador irá utilizar o destrutor da classe-base, liberando assim a memória alocada.

16.4 Construtor de Cópia & Alocação Dinâmica

O programa cria o objeto **P2** e então atribui a ele o conteúdo do objeto **P1**. O seguinte problema acontecerá: quando você atribui um objeto a outro, o compilador copia, byte a byte, todo o conteúdo de memória de um objeto no outro. Em outras palavras, ao membro **NOME** de **P2** será atribuído o conteúdo do membro **NOME** de **P1**. Entretanto, o membro **NOME** é um ponteiro e, como resultado, **P2.NOME** e **P1.NOME** apontarão para a mesma localização na memória.

Desta forma, qualquer modificação na cadeia de caracteres de um dos objetos afetará diretamente o outro. Um problema mais sério ocorre quando os objetos envolvidos têm escopos diferentes. Quando o destrutor de **P1** é chamado, destruirá a memória alocada e apontada por **P1**, destruindo portanto a mesma memória apontada por **P2**. Se **P2** tentar acessar o conteúdo deste ponteiro, um erro fatal será gerado.

Para evitar que este erro aconteça, implementamos o construtor de cópia para os objetos da classe **Pessoa**:

```
Pessoa(Pessoa const & aP);
```

Este construtor aloca um espaço de memória para armazenar os seus dados e depois copia os valores contidos do objeto passado como argumento. Note que esta função não copia o endereço contido em **NOME**, mas sim o valor apontado por este ponteiro. Evitando assim o problema descrito acima. Este construtor de cópia é muito importante, geralmente utilizado na conversão de outros

objetos para este tipo ou na atribuição de objetos. Reimplemente este operador sempre que a sua classe contiver ponteiros com alocação dinâmica como atributos.

16.5 Chamadas a Funções

O objeto C1 da classe `Cliente` usa a função `GetData()` membro da classe `Cliente`, na instrução:

```
C1.GetData();
```

O compilador sempre procurará primeiro pela função na classe que define o objeto (no caso, `Cliente`). Se o compilador não encontrar esta função então ele irá procurar nas classes-bases da classe do objeto. Se ele encontrar em uma classe-base, o compilador executará a função encontrada senão ocorrerá um erro.

Podemos criar funções-membro de uma classe derivada que tenham o mesmo nome de funções-membro da classe-base. Isto faz com que a sintaxe da chamada a elas, por meio de um objeto, seja a mesma, independentemente de tratar-se de um objeto da classe-base ou da classe derivada.

A função `GetData()` está definida com o mesmo protótipo tanto na classe-derivada quanto na classe-base, então qual das versões da função será utilizada pelo compilador? A regra é a seguinte: se duas funções de mesmo nome existem, uma na classe-base e outra na classe derivada, a função da classe derivada será executada se for chamada por meio de um objeto da classe derivada. Se um objeto da classe base é criado, usará sempre funções da própria classe-base pois não conhece nada sobre a classe derivada.

Para que uma função da classe derivada, com mesmo nome de uma função da classe-base, possa acessar a que está na classe-base, é necessário o uso do operador de resolução de escopo (`::`):

```
Pessoa::GetData();
```

Esta instrução executa a função `GetData()` da classe-base dentro da definição desta função na classe-derivada. Sem o uso do operador de resolução de escopo, o compilador executará a função `GetData()` da classe-derivada e o resultado será uma seqüência infinita de chamadas recursivas. O operador de resolução de escopo permite que se especifique exatamente qual é a classe da função que queremos executar.

```
C2.GetData();  
C1.Pessoa::GetData();
```

A diferença das instruções acima é que para o primeiro objeto, estaremos utilizando a versão da função `GetData()` definida na classe derivada e para o segundo objeto estaremos utilizando a versão da função definida na classe-base. A distinção entre as duas chamadas é feita pelo operador de resolução de escopo (`::`).

16.6 Herança Pública e Privada

Declaramos a classe derivada `Cliente` especificando a palavra `public`:

```
class Cliente : public Pessoa
```

A declaração `public` indica que os membros públicos da classe-base serão membros públicos da classe derivada e os membros `protected` da classe base serão também membros `protected` da classe derivada. Os membros públicos da classe-base podem ser acessados por um objeto da classe derivada.

A declaração `private` pode ser usada no lugar de `public` e indica que tanto os membros públicos quanto os protegidos da classe-base serão membros privados da classe derivada. Estes membros são acessíveis aos membros da classe derivada, mas não aos seus objetos. Portanto, um objeto da classe derivada não terá acesso a nenhum membro da classe-base.

Os membros privados da classe-base serão sempre inacessíveis fora da classe-base. Nenhuma classe derivada pode acessar um membro privado de uma classe base.

Geralmente, a derivação privada é raramente usada. Mas há momentos em que a derivação privada é desejável. Por exemplo, imagine que você tenha uma função da classe-base que trabalha perfeitamente com objetos da classe base, mas gera resultados errados quando usado com objetos da classe derivada. A derivação privada neste caso é uma solução bem elegante.

16.7 Conversões de Tipos entre Classe-Base e Derivada

Visto que `Cliente` é um tipo de `Pessoa`, faz sentido pensar em converter um objeto da classe `Cliente` num objeto da classe `Pessoa`. C++ permite a conversão implícita de um objeto da classe derivada num objeto da classe-base. Por exemplo:

```
Pessoa P3 = C1; // C1 é um objeto da classe Cliente
```

Todos os membros da classe-base `P3` recebem os valores dos membros correspondentes do objeto `C1` da classe derivada. Aqui, o construtor de cópia da classe `Pessoa` evitará que um erro ocorra devido a um ponteiro presente na classe. Este construtor é utilizado para converter o objeto da classe derivada em um objeto da classe base sem que ambos objetos contenham o mesmo endereço no ponteiro e venham a causar um erro de acesso à memória.

Entretanto a atribuição inversa não é válida, não podemos atribuir um objeto da classe-base a um objeto da classe derivada.

Sobrecarregamos os operadores `<<` associados a objetos da classe `ostream`, para facilitar a apresentação dos dados dos objetos de ambas as classes na tela. Perceba que na implementação da sobrecarga do operador `<<` para o objeto da classe `ostream` e um objeto da classe `Cliente`, que utilizamos uma conversão na instrução abaixo:

```
OS << Pessoa(C);
```

Esta conversão converte temporariamente o objeto `C` da classe `Cliente` para um objeto da classe `Pessoa`. Isto causará a chamada do operador `<<` definido para a classe `Pessoa`, imprimindo na tela os dados da classe `Pessoa` presentes no objeto da classe `Cliente`. Esta instrução opera como se estivéssemos chamando uma função da classe-base utilizando o operador de resolução de escopo. Aqui não utilizamos o operador, mas convertemos o objeto da classe derivada, para um objeto da classe base antes de chamarmos a função, o que tem o mesmo resultado.

16.8 Níveis de Herança

Uma classe por ser derivada de outra classe, que, por sua vez é também uma classe derivada.

```
class X {};  
class Y : public X {};  
class Z : public Y {};
```

Neste exemplo, `Y` é derivada de `X`, e `Z` é derivada de `Y`. Este processo pode ser estendido para qualquer número de níveis, `K` pode ser derivada de `Z`, e assim por diante.

16.9 Herança Múltipla

Uma classe pode herdar as características de mais de uma classe. Este processo é chamado de herança múltipla. A construção de hierarquias de herança múltipla envolve mais complexidade do que as hierarquias de herança simples. Esta complexidade diz respeito ao desenho da construção das classes e não à sintaxe de uso. A sintaxe de múltiplas heranças é similar àquela de uma única herança. Eis um exemplo que representa os imóveis à venda de uma imobiliária :

```
// Heranca Multipla
#include <iostream>
#include <cstdio>
#include <iomanip>
#include <string>
using namespace std;

class Cadastro
{
private:
    char nome[30], fone[20];
public:
    Cadastro() { nome[0] = fone[0] = '\0'; }
    Cadastro(char n[], char f[])
    {
        strcpy(nome, n);
        strcpy(fone, f);
    }
    void GetData()
    {
        cout << "\n\tNome: ";
        cin.getline(nome, 30);
        cout << "\tFone: ";
        cin.getline(fone, 20);
    }
    void PutData()
    {
        cout << "\n\tNome: " << nome;
        cout << "\n\tFone: " << fone;
    }
};

class Imovel
{
private:
    char end[30], bairro[20];
    float AreaUtil, AreaTotal;
    int quartos;
public:
    Imovel()
    {
        end[0] = bairro[0] = '\0';
        AreaUtil = AreaTotal = 0;
        quartos = 0;
    }
    Imovel(char e[], char b[], float au, float at, int q)
    {
```

```
        strcpy(end, e);
        strcpy(bairro,b);
        AreaUtil = au;
        AreaTotal = at;
        quartos = q;
    }
    void GetData()
    {
        cout << "\n\tEnd: "; cin.getline(end, 30);
        cout << "\tBairro: "; cin.getline(bairro, 20);
        cout << "\tArea Util: "; cin >> AreaUtil;
        cout << "\tArea Total: "; cin >> AreaTotal;
        cout << "\tNo. Quartos: "; cin >> quartos;
    }
    void PutData()
    {
        cout << "\n\tEnd : " << end;
        cout << "\n\tBairro : " << bairro;
        cout << "\n\tArea Util : " << setiosflags(ios::fixed)
            << setprecision(2) << AreaUtil
            << "\n\tArea Total : " << AreaTotal
            << "\n\tQuartos : " << quartos;
    }
};

class Tipo
{
private:
    char tipo[20]; // Residencial, Loja, Galpao
public:
    Tipo() { tipo[0] = '\0'; }
    Tipo(char t[]) { strcpy(tipo, t); }

    void GetData()
    {
        cout << "\n\tTipo : ";
        cin.getline(tipo, 20);
    }
    void PutData()
    {
        cout << "\n\tTipo: " << tipo;
    }
};

class Venda : private Cadastro, public Imovel, public Tipo
{
private:
    float valor;
public:
    Venda() : Cadastro(), Imovel(), Tipo() {valor = 0;}
    Venda(char n[], char f[], char e[], char b[],
        float au, float at, int q, char t[], float v)
        : Cadastro(n,f), Imovel(e,b,au,at,q), Tipo(t)
        { valor = v; }
    void GetData()
    {
```

```
        cout << "\n ...Proprietarios: ";
        Cadastro::GetData();
        cout << "\n ...Imovel: ";
        Imovel::GetData();
        Tipo::GetData();
        cout << "\tValor R$ : "; cin >> valor;
    }
    void PutData()
    {
        cout << "\n ...Proprietario: ";
        Cadastro::PutData();
        cout << "\n ...Imovel: ";
        Imovel::PutData();
        Tipo::PutData();
        cout << "\n\tValor R$ : " << valor;
    }
};
int main(int argc, char* argv[])
{
    Venda C,V("Eduardo Mattos", "3203322", "Rua Rocha, 33", "Coqueiros",
              50.0, 75.0, 2, "Comercial", 80000.0 );
    cout << "\n\n* Imovel Para Venda : ";
    V.PutData();
    cout << "\n\n Digite um Imovel para Vender!\n";
    C.GetData();    C.PutData();

    return 0;
}
```

Através da declaração:

```
class Venda : private Cadastro, public Imovel, public Tipo { ... };
```

notificamos o compilador que a classe Venda é derivada das classes Cadastro, Imovel e Tipo. Cadastro tem derivação do tipo privada enquanto que as demais classes têm derivações do tipo públicas. Muitas vezes podemos substituir uma herança por um objeto da classe-base. A escolha de qual método devemos aplicar depende muito do modelo adotado e do desempenho esperado. Uma boa modelagem pode definir claramente quais serão as classes criadas e como elas deverão iteragir.

Os construtores da classe Venda são os seguintes:

```
Venda() : Cadastro(), Imovel(), Tipo() {valor = 0;}
Venda(char n[], char f[], char e[], char b[],
      float au, float at, int q, char t[], float v)
      : Cadastro(n,f), Imovel(e,b,au,at,q), Tipo(t)
      { valor = v; }
```

Note que a definição de um construtor de uma classe com herança múltipla é muito similar ao construtor de uma classe com herança simples. Ao chamar os construtores das classes-bases, devemos lembrar que os nomes dos construtores são colocados após os dois-pontos e separados por vírgulas. A ordem de chamada é a mesma da ordem em que eles aparecem escritos. Parâmetros podem ser passados assim como em qualquer outro construtor.

As funções GetData() e PutData() da classe Venda incorporam a chamada às funções correspondentes das classes Cadastro, Imovel e Tipo empregando o operador de resolução de escopo (::) para efetuar a chamada das funções das classes base:

```
Cadastro::GetData(); Imovel::GetData(); Tipo::GetData();  
Cadastro::PutData(); Imovel::PutData(); Tipo::PutData();
```

Alguns tipos de problemas podem aparecer em certas situações envolvendo herança múltipla. O mais comum é quando duas classes-base têm, cada uma delas, uma função de mesmo nome (protótipo), enquanto a classe derivada destas duas não tem nenhuma função com este nome. Quando a função é acessada por meio de um objeto da classe derivada, o compilador não reconhecerá qual das duas estará sendo chamada. Para informar ao compilador qual das duas funções está sendo solicitada, devemos utilizar o operador de resolução de escopo (::). Por exemplo, podemos chamar a função PutData() da classe Imovel com um objeto da classe Venda através da instrução:

```
C.Imovel::PutData();
```

Caso a classe derivada não contivesse esta função, este seria o único método de acesso a esta função sem causar um erro de ambiguidade.

16.10 Exercícios

16.1 No início começamos discutindo a modelagem de um banco orientado a objetos. Termine o programa que modela as pessoas que compõem o ambiente banco : acionista, cliente, funcionário, cliente-universitário, pessoa-física, pessoa-jurídica, etc.

16.2 Crie uma classe Empresa capaz de armazenar os dados de uma empresa (Nome, End, Cidade, Estado, CEP, CGC, Fone e E-mail). Use a classe Empresa como base para criar a classe Restaurante. Inclua o tipo de comida, o preço médio de um prato, funções para adquirir os seus dados e para imprimí-los na tela. Sobrecarrega o operador ostream <<.

16.3 Imagine que você deva escrever um programa para armazenar veículos. Primeiramente, crie a classe Motor que contém NumCilindro e Potencia. Inclua um construtor com argumentos com valores default que inicialize os dados com zeros e um construtor de cópia. Escreva a classe Veículo contendo peso, velocidade máxima e preço. Crie a classe CarroPasseio usando as classes Motor e Veículo como base. Inclua cor e modelo. Crie a classe caminhão derivada das classes Motor e Veículo. Inclua a carga máxima, altura máxima e comprimento. Crie um programa para o controle de um estacionamento que tenha uma área fixa e controle o acesso de veículos no estacionamento. Verifique se um veículo pode ou não estacionar antes de permitir a entrada. Se puder, indique onde este deve estacionar. Reflita sobre o problema, crie um modelo e decida quais propriedades dos veículos devem ser descritas. Aceite as propriedades propostas como sugestões mas crie o seu próprio programa.

16.4 Um robô pode ser composto por três tipos de motores : motor de corrente contínua, motor de passo e motor assíncrono. Um robô pode conter juntas com movimento angular (geram uma variação no ângulo entre dois elos), transversal (geram um deslocamento linear dos elos) ou juntas que não geram movimento nenhum, simplesmente ligam dois elos. Cada elô pode ser uma barra reta ou uma curva suave de 90 graus. Um robô pode ter uma garra para pegar objetos, uma garra para fazer operações de soldagem e uma garra para fazer operações de pintura. Modele um robô segundo as considerações acima, definindo a geometria do robô e como será o movimento deste. Crie um robô que seja capaz de pintar a parte superior de uma mesa, soldar em cima desta uma caixa. A mesa esta a 50 cm na frente do robô e a caixa está a 50 cm à direita do robô.

17 Funções Virtuais e Amigas

Neste capítulo discutiremos tópicos avançados sobre classes. Cobriremos a definição e o uso de funções virtuais e funções amigas.

17.1 Funções Virtuais

Muitas vezes, quando derivamos um grupo de classes a partir de uma classe-base, certas funções-membro precisam ser redefinidas em cada uma das classes derivadas. Suponha o seguinte exemplo, onde temos uma classe base e duas classes derivadas. A função `print()` está definida na classe base e redefinida nas classes derivadas. As nossas classes definem os tipos de clientes em um banco. Temos os clientes normais (classe-base) e clientes universitários ou especiais.

```
// Testando funcoes Virtuais
#include <iostream>
using namespace std;
class Cliente {
public:
    virtual void print() { cout << "\nCliente"; }
};
class Universitario : public Cliente {
public:
    void print() { cout << "\nCliente Universitario"; }
};
class Especial : public Cliente {
public:
    void print() { cout << "\nCliente Especial"; }
};
int main(int argc, char* argv[])
{
    Cliente * p[3]; // matriz de ponteiros da classe-base
    Cliente B;
    Universitario D1;
    Especial D2;
    p[0] = &B;
    p[1] = &D1;
    p[2] = &D2;
    p[0]->print();
    p[1]->print();
    p[2]->print();

    return 0;
}
```

Planejamos agrupar um conjunto de objetos destas classes em uma lista única. Isto é, o nosso objetivo é ter uma lista única de clientes e diferenciá-los pelos tipos de objetos que os definem. Um dos meios é criar uma matriz de ponteiros para os diferentes objetos envolvidos. Qual seria o tipo da matriz tendo em vista que ela deve armazenar ponteiros para objetos de classes diferentes? A resposta está em como o compilador opera ponteiros em situações deste tipo. Um ponteiro para um objeto de uma classe derivada é de tipo compatível com um ponteiro para um objeto da classe-base. Segundo este princípio, criamos um vetor de ponteiros da classe-base e atribuímos a ele endereços das classes derivadas, como você pode observar acima.

Podemos imprimir os dados da lista, através da chamada da função `print()` :

```
for(int i = 0; i<10; i++) p[i]->print();
```

Observe que desejamos que funções completamente diferentes sejam executadas por meio da mesma instrução de chamada. Se o ponteiro `p[i]` apontar para um cliente normal, a função da classe base deverá ser executada; se `p[i]` apontar para um cliente especial, a função da classe de clientes especiais deverá ser executada.

A característica de chamar funções membros de um objeto sem especificar o tipo exato do objeto é conhecida como polimorfismo. A palavra polimorfismo significa “assumir várias formas”. Em C++ indica a habilidade de uma única instrução chamar diferentes funções e portanto assumir diferentes formas.

Para que o exemplo anterior funcionasse corretamente, duas condições tiveram que ser introduzidas. Em primeiro lugar, todas as diferentes classes de clientes devem ser derivadas da mesma classe-base. Em segundo lugar, a função `print()` deve ser declarada como virtual na classe-base.

Apesar de termos fornecidos os endereços de objetos de classes derivadas, o compilador trata todos os endereços da lista de ponteiros como endereços da classe-base, ou seja, considera que os ponteiros apontam para objetos da classe-base. Quando executamos uma função utilizando este ponteiro,

```
p[i]->print();
```

a função será executada como se o objeto referenciado fosse da classe base.

```
p[i]->Cliente::print();
```

o compilador ignora o conteúdo do ponteiro `p[i]` e usa o seu tipo para identificar a função-membro a ser chamada. Para acessar objetos de diferentes classes usando a mesma instrução, devemos declarar as funções da classe-base que serão reescritas em classes derivadas usando a palavra-chave `virtual`.

As instruções de chamada a funções não-virtuais são resolvidas em tempo de compilação e são traduzidas em chamadas a funções de endereços fixos. Isto faz com que a instrução seja vinculada à função antes da execução. Quando uma instrução de chamada a uma função virtual é encontrada pelo compilador, ele não tem como identificar qual é a função associada em tempo de compilação. Em instruções como a escrita acima, o compilador não conhece qual classe `p[i]` contém antes de o programa ser executado. Então, a instrução é avaliada em tempo de execução, quando é possível identificar qual o tipo de objeto é apontado por `p[i]`. Isto é chamado de resolução dinâmica.

A resolução dinâmica permite que uma instrução seja associada a uma função no momento de sua execução. O programador especifica que uma determinada ação deve ser tomada em um objeto por meio de uma instrução. O programa, na hora da execução, interpreta a ação e vincula a ação à função apropriada. A resolução dinâmica envolve mais memória e tempo de execução, entretanto aumenta a flexibilidade no projeto do software e permite criar bibliotecas de classes que outros programadores podem estender não tendo o arquivo-fonte.

Uma função virtual pura é uma função virtual sem bloco de código ou o bloco de código não contém nenhuma instrução. O propósito de uso de uma função virtual pura é em situações em que a função nunca será executada e está presente somente para que seja redefinida em todas as classes derivadas. A função serve somente para prover uma interface polimórfica para as classes derivadas. A função `print()` pode ser definida como virtual pura da seguinte forma:

```
virtual void print() =0;
```

O sinal de igual (=) e o valor 0 não têm nenhum efeito aqui. A sintaxe (=0) é usada somente para indicar ao compilador que esta é uma função virtual pura, e não tem corpo.

Uma classe que não pode ser utilizada para criar objetos é dita classe abstrata e existe somente para ser usada como base para outras classes. A classe que define uma função virtual pura é uma classe abstrata pois não se pode declarar nenhum objeto dela. Entretanto, um ponteiro para uma classe abstrata pode ser declarado para manipular objetos de classes derivadas. Toda classe que contém pelo menos uma função virtual pura ou uma classe derivada que não redefine a função virtual pura de sua classe-base é abstrata.

17.2 Destrutores Virtuais

Se destrutores são definidos na classe-base e na classe derivada, eles são executados na ordem reversa à qual o construtor é executado. Quando um objeto da classe derivada é destruído, o destrutor da classe derivada é chamado e em seguida o destrutor da classe-base é chamado.

Esta ordem não é mantida quando um ponteiro para a classe-base contém um objeto de uma classe derivada. Se o operador delete é aplicado a um ponteiro da classe-base, o compilador chama somente o destrutor da classe-base, mesmo que o ponteiro aponte para um objeto da classe derivada.

A solução é declarar o destrutor da classe-base virtual. Isto faz com que os destrutores de todas as classes derivadas sejam virtuais, mesmo que não compartilhem o mesmo nome do destrutor da classe-base. Então, se delete é aplicado a um ponteiro para a classe-base, os destrutores apropriados são chamados, independentemente de qual tipo de objeto é apontado pelo ponteiro.

```
class Base
{
    public:    virtual ~Base(); // destrutor virtual
};
```

A classe Base tem um destrutor virtual, mesmo sendo um destrutor que não faz nada. Sempre que você escrever uma classe que tem uma função virtual, ela deve ter um destrutor virtual mesmo que não necessite dele. A razão disto é que uma classe derivada pode necessitar de um destrutor. Se um destrutor virtual está definido na classe-base, você assegura que destrutores de classes derivadas são chamados na ordem necessária. Observe que, enquanto os destrutores podem ser virtuais, os construtores não devem ser.

17.3 Classe-Base Virtual

Além de declarar funções virtuais, podemos usar a palavra virtual para declarar uma classe inteira. A necessidade de declarar uma classe virtual é quando esta foi usada como classe-base para mais de uma classe derivada e dessas classes derivadas foi derivada outra por meio de herança múltipla. Uma classe-base não pode ser usada mais de uma vez numa mesma classe derivada. Entretanto, uma classe-base pode ser indiretamente usada mais de uma vez em classes derivadas. Observe a classe Super do exemplo abaixo:

```
#include <iostream>
using namespace std;

class Base {
    protected: int valor;
    public:
        Base(int n = 0) {valor = n;}
        virtual void Print() { cout << "\nValor : " <<
            valor; }
};

class Deriv1 : virtual public Base {
    public:
        Deriv1(int n = 0) : Base(n) { }
```

```
};
class Deriv2 : virtual public Base {
public:
    Deriv2(int n = 0) : Base(n) { }
};
class Super : public Deriv1, public Deriv2
{
    public:
    Super(int n = 0) : Base(n) { }
    int RetValor() { return valor; }
    void Print() { cout << "\nValor do Super-Objeto : " << valor; }
};
int main(int argc, char* argv[])
{
    Base B(5);
    Deriv1 D1(10);
    Super S(343);
    B.Print();
    D1.Print();
    S.Print();

    return 0;
}
```

Neste caso, cada objeto da class Super poderia ter dois sub-objetos da classe Base. Se um objeto da classe Super tentasse acessar dados ou funções da classe Base, ocorreria um erro de ambiguidade. A declaração virtual nas classes Deriv1 e Deriv2 faz com que estas classes compartilhem uma única classe-base. A classe Super só terá um subobjeto da classe Base e não terá mais nenhum problema de ambigüidade.

Os construtores de classes-base são sempre executados antes do construtor da classe derivada. Os construtores de classes-base virtuais são chamados antes de qualquer construtor de classes-bases não-virtuais.

17.4 Funções Amigas

O conceito de isolamento interno dos itens de uma classe, onde funções não-membros não teriam o privilégio de acesso aos dados privados ou protegidos desta classe, é violado por um mecanismo oferecido por C++ que permite que funções não-membro, às quais foi dada uma permissão especial, tenham acesso à parte interna da classe. Declarando uma função como amiga, ela tem os mesmos privilégios de uma função-membro, mas não está associada a um objeto da classe. No programa abaixo apresentamos a sintaxe das funções amigas:

```
// Uso de funcoes amigas
#include <iostream>
#include <strstrea>
using namespace std;

class Data; // Declara que existe esta classe

class Tempo
{
private:
    long h, min, s;
public:
    Tempo(long hh = 0, long mm = 0, long ss = 0)
    { h = hh; min = mm; s = ss; }
```

```
friend char * PrintTime( Tempo &, Data &);
};
class Data
{
private:
    int d, m, a;
public:
    Data(int dd = 0, int mm = 0, int aa = 0)
    { d = dd; m = mm; a = aa; }
    friend char * PrintTime( Tempo &, Data &);
};
char * PrintTime( Tempo & Tm, Data & Dt) // Função global e amiga
{
    char * temp = new char[50];
    memset(temp, '\0', 50);
    stringstream sIO(temp, 50, ios::out);
    sIO << "\n Relógio-> \t" << Tm.h << ":" << Tm.min << ":" << Tm.s;
    sIO << "\n Data-> \t" << Dt.d << "/" << Dt.m << "/" << Dt.a;
    return temp;
}
int main(int argc, char* argv[])
{
    Tempo Tm( 15, 56, 4);
    Data Dt( 9, 5, 2000);
    char * str = PrintTime( Tm, Dt);
    cout << "\n" << str;
    delete str;

    return 0;
}
```

Neste exemplo, queremos que a função `PrintTime()` tenha acesso aos dados privados da classe `Tempo` e `Data`. Desta forma, usamos a palavra-chave `friend` na declaração da função `friend char * PrintTime( Tempo &, Data &);`

Esta declaração pode ser colocada em qualquer posição da classe. Não há diferença se for colocada na parte pública ou na parte privada da classe. Um objeto `Tempo` é passado como argumento à função `PrintTime()`. Este argumento é necessário pois, mesmo quando se dá a uma função amiga o privilégio de acesso aos dados privados da classe, ela não é parte daquela classe, devendo ser informada sobre qual objeto agirá.

A função `PrintTime()` cria um objeto da classe `stringstream`. Este objeto atua como se fosse um objeto `cout`, entretanto em vez de enviar os dados para a saída padrão (tela), ele armazena estes dados no buffer de caracteres fornecidos ao seu construtor. Esta é uma das formas mais fáceis para converter dados de uma classe para texto (ASCII) e armazenar estes dados em um buffer de caracteres. Note que operamos sob o objeto criado assim como operamos com o objeto `cout`. O mesmo princípio pode ser empregado para a leitura de dados armazenados em um buffer de caracteres, operando-se como se estivessemos utilizando o operador `cin`.

Funções amigas são muito empregadas quando sobrecarregamos os operadores `<<` e `>>` associados aos objetos `cout` e `cin` respectivamente. Desta forma definimos o operador `<<` para imprimir os dados da classe e o operador `>>` para adquirir os dados da classe. Como, muitas vezes desejamos que estes operadores tenham acesso aos membros privados da classe, então devemos defini-los como funções amigas.

```
friend ostream & operator <<(ostream & OS, Tempo const & T);
```

```
friend istream & operator >>(istream & IS, Tempo & T;
```

Todo e qualquer operador que queremos sobrecarregar como sendo uma função global (i.e., recebe um parâmetro se for unário e dois se for binário) e queremos permitir que este operador tenha acesso a membros privados e protegidos de uma dada classe, devemos definir este operador como sendo uma função amiga.

No nosso programa exemplo acima, definimos que a função `PrintTime()` é amiga das classes `Data` e `Tempo`. Para tal, precisamos declarar nas duas classes que esta função é amiga da classe. Quando declaramos a função na classe `Tempo`, estaremos declarando uma função que recebe como parâmetro um objeto da classe `Data` desconhecida até então pelo compilador. Para evitar que o compilador gere um erro indicando que a classe não existe, devemos declarar no início do arquivo que existe a classe `Data`, e que esta será definida posteriormente. Fazemos isto através da instrução:

```
class Data;
```

17.5 Classes Amigas

Além de ser possível declarar funções independentes como amigas, podemos declarar uma classe toda como amiga de outra. Neste caso, as funções-membro da classe serão todas amigas da outra classe. A diferença é que estas funções-membro têm também acesso à parte privada ou protegida da sua própria classe.

Para modificar o programa anterior e declarar que a classe `Data` é amiga da classe `Tempo`, basta retirarmos a declaração da função amiga `PrintTime()` da classe `Tempo` e adicionarmos a declaração:

```
friend class Data;
```

Agora, se definirmos que a função `PrintTime()` passa a ser uma função da classe `Data`, ela automaticamente passará a ser amiga da classe `Tempo` e terá acesso a todos os membros desta classe.

Uma função ou classe amiga deve ser declarada como tal dentro da classe em que os dados serão acessados. Em outras palavras, quem escreve uma classe deve especificar quais funções ou classes irão acessar a sua parte privada ou protegida. Desta forma, um programador que não tem acesso ao código-fonte da classe não poderá criar uma função amiga para esta classe. As funções amigas devem ser declaradas pelo autor da classe. Observe que a palavra chave `friend` oferece acesso em uma só direção. Se a classe `A` é amiga da classe `B`, o inverso não é verdadeiro.

Uma função operadora amiga deve receber no mínimo um argumento objeto. Em outras palavras, você não pode criar uma função operadora binária como operador `+` que receba dois inteiros como argumentos. Esta restrição previne a redefinição de operadores internos da linguagem. Alguns operadores não podem ser definidos como funções amigas; devem ser definidos como funções-membro de classe. São eles: atribuição (`=`), acesso de membros de ponteiros (`->`), `()` e conversores de tipo.

17.6 Exercícios

17.1 Escreva as classes `Animal`, `Vaca`, `Bufalo` e `Bezerro`, sendo `Vaca` e `Bufalo` derivadas de `Animal` e `Bezerro` derivada de `Vaca` e de `Bufalo`.

17.2 Modele os elementos `Veículos`, `Moto`, `Carro Esportivo`, `Carro de Luxo`, `Carro Popular`, `Perua`, `Carro Mil`, `Jipe`, `Caminhão`, `Onibus` e `Caminhonete`. Utilize a idéia do exercício 16.3. Crie um programa que controle uma revenda de carros. Tenha uma lista única contendo todos os veículos a

venda. Utilize o princípio de um vetor de ponteiros para a classe base e empregue exhaustivamente os conceitos da derivação e de funções virtuais e amigas. Permita que o usuário insira algum veículo na lista, faça consultas, liste todos os veículos disponíveis e retire um veículo da lista.

17.3 Defina uma classe String que sobrecarrega os seguintes operadores para as mais variadas funções: (+), (==), (!=), (<), (>), (<=), (>=), (<<) e (>>).

17.4 Refaça os exercícios 16.1 e 16.2 empregando o princípios das funções amigas e virtuais.

18 Operações com Arquivos Iostream

Neste capítulo, veremos brevemente as facilidades de C++ para operações de leitura e gravação em discos. Começaremos com uma breve descrição das classes usadas para executar esta tarefa, conhecidas como classes iostream.

Em C++, os objetos stream são o ponto central das classes iostream. Um objeto stream pode ser pensado como um buffer para o recebimento ou envio de bytes. Desta forma, um arquivo em disco é um objeto stream. O conceito de arquivo é ampliado no sentido de considerar arquivos não somente os que existem em discos mas também o teclado, o vídeo, a impressora e as portas de comunicação. Por exemplo, o objeto cout representa o vídeo (recebimento de bytes) e o objeto cin representa o teclado (envio de bytes). As classes iostream interagem com esses arquivos.

Diferentes objetos stream são usados para representar diferentes interações com periféricos de entrada e saída. Cada stream é associado a uma classe particular, caracterizada pelas suas funções de interface e pelos seus operadores de inserção e extração.

Na Figura 4 apresentamos a hierarquia das classes iostream. Por exemplo, cout é um objeto da classe ostream_withassign que é derivada da classe ostream. Similarmente, cin é um objeto da classe istream_withassign que é derivada da classe istream. Objetos de destino de bytes usam o operador de inserção <<, membro da classe ostream. Os objetos de origem de bytes usam o operador de extração >>, membro da classe istream. Estas duas classes são derivadas da classe ios. Para descobrir os arquivos que definem estas classes, utilize o help do seu compilador pois a nova norma ANSI C++ definiu novos nomes para estes headers.

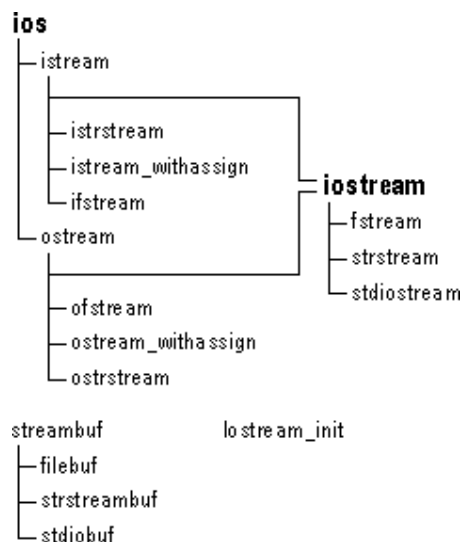


Figura 4 : Hierarquia das Classes Iostream do ANSI C++ obtidas pelo Help do Visual C++ 6.0.

A classe ios é a base para todas as outras classes. Ela contém várias constantes e funções-membro para as operações de leitura e impressão a todas as outras classes.

As classes istream e ostream são derivadas de ios e são dedicadas a leitura e impressão, respectivamente. Suas funções-membro implementam operações formatadas ou não-formatadas em objetos stream. A classe istream contém funções como get(), getline(), read(), além de outras. Contém ainda a sobrecarga do operador de extração >>. Já a classe ostream contém funções como put(), write(), além de outras. Contém ainda a sobrecarga do operador de inserção <<. Todo objeto istream ou ostream contém um objeto streambuf derivado. As classes istream e ostream trabalham em conjunto com a classe streambuf.

A classe `iostream` é derivada das classes `istream` e `ostream` por meio de herança múltipla. Desta forma, esta classe herda as capacidades tanto de leitura como de impressão. Outras classes podem ser criadas pelo usuário tendo como base `istream` e `ostream`.

As classes `istream_withassign` e `ostream_withassign` são usadas para a entrada e a saída padrão. Os objetos `cin` e `cout` são objetos destas classes.

As classes `istrstream`, `ostrstream` e `strstream` são usadas na manipulação de buffers de dados: para leitura, para gravação e para leitura/gravação respectivamente.

As classes `ifstream`, `ofstream` e `fstream` são usadas na manipulação de arquivos em disco para leitura, para gravação e para leitura e gravação respectivamente.

18.1 Estudo de Caso

Vamos apresentar a leitura e escrita de arquivos a partir de um exemplo. O exemplo tratado é a classe `Pessoa` definida nos capítulos anteriores. A classe em si não apresenta nada de novo, somente sobrecarregamos os operadores de inserção `<<` e extração `>>` de dados definidos para objetos da classe `iostream`. A sobrecarga destes operadores nos permite a escrita e a leitura automática de dados da classe, utilizando-se estes operadores. O header `fstream.h` foi adicionado por causa das classes de manipulação de arquivos. Veja o exemplo abaixo.

```
// Exemplo da class fostream
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
using namespace std;

#define Nome_Size 80

class Pessoa
{
protected:
    char *      Nome;
    int         Idade;
    long int    RG;

public:
    Pessoa();
    Pessoa(char * aNome, int aId = 0, long int aRG = 0);
    Pessoa(Pessoa const & aP);
    ~Pessoa();

    char const * GetNome() const { return Nome; }
    int         GetIdade() const { return Idade; }
    long int     GetRG() const { return RG; }

    void         GetData();

    friend ostream & operator <<(ostream & OS, Pessoa const & P);
    friend istream & operator >>(istream & IS, Pessoa & P);
};

Pessoa::Pessoa()
{
    Nome = new char[Nome_Size];
```



```
        Idade = 0;
        RG = 0;
    }

Pessoa::Pessoa( char * aNome, int aId, long int aRG)
{
    Nome = new char[Nome_Size];
    strcpy( Nome, aNome);
    Idade = aId;
    RG = aRG;
}

Pessoa::Pessoa(Pessoa const & aP)
{
    Nome = new char[Nome_Size];
    strcpy( Nome, aP.GetNome());
    Idade = aP.GetIdade();
    RG = aP.GetRG();
}

Pessoa::~Pessoa()
{
    if(Nome != 0)
    {
        delete Nome;
        Nome = 0;
    }
}

void Pessoa::GetData()
{
    cout << "\nNome : ";
    cin.getline( (char *)Nome, Nome_Size);
    cout << "Idade : ";
    cin >> Idade;
    cout << "RG : ";
    cin >> RG;
}

ostream & operator <<(ostream & OS, Pessoa const & P)
{
    OS    << P.GetNome()
    << "\n" << P.GetIdade()
    << "\n" << P.GetRG();
    return OS;
}

istream & operator >>(istream & IS, Pessoa & P)
{
    IS.getline( (char *)P.Nome, Nome_Size);
    IS >> P.Idade;
    IS >> P.RG;
    return IS;
}
```

```
int main(int argc, char* argv[])
{
    Pessoa Eu("Rodrigo", 20, 231123);

    // Escrevendo
    ofstream FOut("Teste.TXT", ios::out);
    FOut << Eu << "\n Um dia a casa cai! \n "
        << setw(12) << setprecision(3) << 12.2345;
    FOut.close();

    // Lendo
    ifstream FIn;
    FIn.open("Teste.TXT", ios::in)
    Pessoa Vc;
    char Buffer[Nome_Size];
    FIn >> Vc;
    cout << "\nVoce: \n" << Vc;
    while(FIn) //Enquanto nao acabar o arquivo
    {
        FIn.getline( Buffer, Nome_Size);
        cout << "\nBuffer : " << Buffer;
    }
    FIn.close();

    Return 0;
}
```

Na função main() é que aparecem as novidades. Primeiro criamos um objeto Pessoa. Depois escrevemos este objeto e outros dados em um arquivo e depois prosseguimos com a leitura deste.

Neste programa definimos um objeto chamado Fout da classe ofstream. Inicializamos este objeto com o nome do arquivo “TESTE.TXT”. Esta inicialização associa o objeto Fout ao arquivo em disco “TESTE.TXT” para gravação e já abre o arquivo. O segundo parâmetro indica que o arquivo será aberto para a escrita.

Note que escrevemos os dados no arquivo da mesma forma que escrevemos os dados na tela com cout. Toda formatação e função válida para cout também pode ser aplicada a FOut. Desta forma, utilizamos o mesmo operador definido para escrever os dados da classe Pessoa na tela como em um arquivo. Esta facilidade advém do fato de cout e FOut serem ambos objetos de classes derivadas da classe ostream.

Após a escrita dos dados, devemos fechar o arquivo com a função close(). É neste momento que os dados serão salvos no arquivo. Se você não fechar o arquivo, os dados somente serão salvos quando o objeto for destruído e o arquivo, então, será fechado pelo destrutor do objeto.

Para ler o arquivo, criamos um objeto da classe ifstream de nome Fin e associamos este objeto ao arquivo Teste.TXT através da função open(). O segundo parâmetro indica que este arquivo será aberto para a leitura. Note que o construtor não abrirá nenhum arquivo aqui, somente após a chamada a função open() é que o arquivo estará aberto.

Depois utilizamos a sobrecarga do operador >> para fazer a aquisição dos dados da classe Pessoa contido no referido arquivo e imprimir na tela. Após termos lidos o dado da classe, prosseguimos lendo linha por linha do arquivo e escrevendo-a na tela até terminar o arquivo. Note que fazemos uso da função getline(), que recebe um buffer como parâmetro e adquire uma linha inteira do arquivo.

Os objetos da classe ifstream tem um valor que pode ser testado para a verificação do fim-de-arquivo. O objeto FIn terá o valor zero se o sistema operacional enviar ao program o sinal de

fim-de-arquivo e um valor não-zero caso contrário. O programa verifica o término do arquivo no laço while. A função eof() desta classe poderia ser empregada com o mesmo propósito.

Após lermos o arquivos, fechamos este através da função close().

Todas as funções definidas para a leitura e escrita em C podem ser empregadas em C++. Estas funções foram incorporadas nas classes istream. Verifique com o help do seu compilador as funções disponíveis e verificará esta observação. Por exemplo, o função put() foi adicionada a classe ostream, e pode ser acessada da forma:

```
cout.put('c');          ou          FOut.put('c');
```

O controle do formato da impressão e leitura pode ser executado por meio dos manipuladores e flags da classe ios. Estes mecanismos foram apresentados em 13.4 e se aplicam para todos os objetos streams.

18.2 A função Open()

Quando usamos um objeto da classe ofstream ou ifstream, é necessário associá-lo a um arquivo. Esta associação pode ser feita pelo construtor da classe ou usando a função open(), membro da classe fstream, numa instrução após a criação do objeto com o construtor default.

Tanto o construtor como a função open() aceitam a inclusão de um segundo argumento indicando o modo de abertura do arquivo. Este modo é definido por bits de um byte, onde cada um especifica um certo aspecto de abertura do arquivo. A lista de modos é definida na classe ios por meio de enum open_mode:

Modos	Descrição
ios::in	Abre para leitura (default de ifstream)
ios::out	Abre para gravação (default de ofstream)
ios::ate	Abre e posiciona no final do arquivo – leitura e gravação
ios::app	Grava a partir do fim do arquivo
ios::trunc	Abre e apaga todo o conteúdo do arquivo
ios::nocreate	Erro de abertura se o arquivo não existir
ios::noreplace	Erro de abertura se o arquivo existir.
ios::binary	Abre em binário (default é texto).

Podemos trabalhar com arquivos no modo texto ou no modo binário, assim como em C. Lembre-se que o fim de linha em modo texto é representado por um único caractere, enquanto em modo binário é representado por dois caracteres.

Exemplos:

```
ifstream fin("Teste.txt", ios::in|ios::binary);  
fstream fio; \\ Arquivo para leitura e gravação!!!!  
fio.open("Lista.dat", ios::in|ios::out|ios::ate);
```

18.3 Testando Erros

Situações de erros na manipulação de arquivos podem ser previstas verificando-se a palavra (int) de status da classe ios. A palavra de status pode ser obtida pela função rdstate(), membro da classe ios. Os bits individuais do valor retornado podem ser testados pelo operador AND bit-a-bit (&) e os seguintes valores enumerados:

Bits	Função	Comentário
ios::goodbit	good()	Nenhum bit setado. Sem erros

ios::eofbit	eof()	Encontrado o fim-de-arquivo
ios::failbit	fail()	Erro de leitura ou gravação
ios::badbit	bad()	Erro irrecuperável

A função `clear(int status)` modifica a palavra de status. Se usada sem argumento, todos os bits de erros são limpos. Do contrário, os bits setados de acordo com os valores enumerados escolhidos e combinados pelo operador OR (|):

```
clear(ios::eofbit | ios::failbit);
```

18.4 Escrevendo e Lendo em Buffers de Caracteres

As classes `iostream` interagem com a memória pela mesma sintaxe com a qual interagem com arquivos de disco. Podemos ler e grava de e para buffers de caracteres na memória do computador. Veja o exemplo abaixo, onde criamos um objeto `ostrstream` para criar um buffer e adicionar dados neste buffer. Depois armazenamos o ponteiro deste buffer em uma variável. Em seguida, criamos um objeto `istrstream` para ler dados do mesmo, passando para o seu construtor o ponteiro do buffer. Por fim, imprimimos os dados na tela.

```
// StrStream
#include <iostream>
#include <strstream>
using namespace std;

int main(int argc, char* argv[])
{
    ostrstream OS; // Cria buffer para a escrita
    OS << "123.45 \t" << 555.55 << "\t" << 333;
    OS << "\n\n Sorte = Estar_Preparado + Oportunidade!";
    OS << ends;

    char * ptr = OS.str();

    double x, y;
    int i;
    istrstream IS(ptr);
    IS >> x >> y >> i;

    cout << x << '\t' << y << '\t' << i << '\n' << ptr;

    return 0;
}
```

Note a semelhança da metodologia. O manipulador `ends`, usando neste exemplo, adiciona o terminador `'\0'` à cadeia de caracteres.

Estas classes podem ser usadas para aproveitarmos os operadores `<<` e `>>` para converter qualquer tipo de dado no formato texto ASCII e armazenar em um buffer de caracteres. Este buffer pode ser utilizado para transmitir estes dados para a tela do micro, ou para uma porta COM ou até para um outro processo via conexões TCP/IP da internet. Definindo o operador `<<` e `>>` para toda classe, definimos um modo padrão para a escrita/leitura dos dados de uma classe para um arquivo, um buffer, ou um canal de I/O qualquer. Esta é uma ferramenta muito poderosa que não pode ser desprezada e, quando bem aproveitada, gera código velozes, simples e com alta flexibilidade.

18.5 Imprimindo em Periféricos

Imprimir dados na impressora é exatamente a mesma coisa que gravar dados num arquivo de disco. O DOS define o nome de alguns periféricos ligados ao computador. Qualquer um deles pode ser usado em nossos programas como nomes de arquivos. Basta criar um objeto do tipo `fstream` (`ifstream`, `ofstream`, `fstream`) e passar como nome do arquivo o nome do periférico. Depois, toda operação feita sob este objeto, irá acessar diretamente o periférico. A tabela seguinte descreve estes nomes:

Nome	Descrição
CON	Console (teclado e vídeo)
AUX ou COM1	Primeira porta serial
COM2	Segunda porta serial
PRN ou LPT1	Primeira porta paralela
LPT2	Segunda porta paralela
LPT3	Terceira porta paralela
NUL	Periférico Inexistente

Exemplo:

```
ofstream oprn("PRN");  
ofstream ocom1("COM1");  
ifstream ilpt2("LPT2");
```

18.6 Exercícios

18.1 Refaça os exercícios 14.1, 14.2 e 14.3, salvando os dados em um arquivo. Utilize todos os conceitos apresentados neste capítulo.

18.2 Escreva um programa para banco de dados de uma biblioteca, que tenha uma lista ligada de livros e estes dados sejam armazenados em um arquivo. Crie uma função para retirar um livro do cadastro e apagá-lo do arquivo.

18.3 Escreva um programa que continue o anterior e pegue os dados de um livro deste cadastro, escreva estes dados em um buffer da memória e depois envie este buffer para ser impresso na impressora.

18.4 Escreva um programa que imprima os dados de um arquivo na tela, 20 linhas por 20 linhas.

18.5 Escreva um programa que converta arquivos criados no modo binário em arquivos criados no modo ASCII. Crie outro programa que faça o contrário.

18.6 Escreva um programa que imprima o tamanho de um arquivo em bytes. O nome do arquivo deve ser fornecido na linha de comando.

18.7 Escreva um programa que criptografa um arquivo usando o operador complemento bit-a-bit (~). Quando o programa é executado para um arquivo já criptografado, o arquivo é recomposto e volta ao original.

19 Namespaces

Namespace é um mecanismo para definir grupos lógicos. Se alguma declaração possuir algo em comum com um outra dada declaração, nós podemos colocá-las juntas em grupo lógico, ou namespace. Estes grupos lógicos são criados para evitar que em um software apareçam duas classes, variáveis ou quaisquer tipos e funções com o mesmo nome. Assim, estes grupos são tratados como um escopo, onde todas as variáveis, tipos e funções definidas neste escopo, são somente conhecidas no seu interior. Se, em outra parte do programa aparecer uma outra variável com o mesmo nome, estas serão diferenciadas por seu escopo, ou seja, pelo namespace (grupo lógico).

19.1 Exemplo e Sintaxe

Um namespace é um escopo e deve ser usado para criar separações lógicas em grandes programas. Como este é um escopo, as regras básicas de escopo são estendidas para o seu caso. Veja o exemplo:

```
namespace RobotController
{
    class RobotSensor
    {
        //declarations and some implementation
    };

    class RobotDCMotors
    {
        //declarations and some implementation
    };

    class RobotMovimentAlgorithm
    {
        //declarations and some implementation
    };
}

namespace RobotVision
{
    class RobotEye
    {
        //declarations and some implementation
    };

    class RobotImageProcessing
    {
        //declarations and some implementation
    };
}
```

Para tornar o código claro, nós usualmente declaramos os namespaces e o seu conteúdo, entretanto, a implementação deste é realizada em outro lugar (arquivo cpp). Não podemos esquecer de qualificar os nomes declarados no namespace com o nome do namespace que o contém:

```
Function_type Namespace_Name::function_name(arguments)
{
    //declarations and some implementation
}
```

A sintaxe para a declaração de um namespace é:

```
namespace Namespace_Name
{
    //declarations and some implementation
}
```

19.2 Qualified Names, Using Declarations and Directives

Quando estamos trabalhando em um específico namespace, podemos usar um nome declarado em outro namespace através da sua qualificação. Veja o exemplo abaixo:

```
double RobotControler::GetPiece(piece*)
{
    //...
    for(//...)
        switch(RobotEye::SquarePiece)//using RobotEye's qualification
    //...
}
```

Se nós estamos trabalhando em um escopo específico, e dentro deste utilizamos constantemente variáveis/funções declarados em outro namespace, nós podemos avisar o compilador que desejamos criar um sinônimo local destas variáveis/funções:

```
double RobotControler::GetPiece(piece*)
{
    using RobotEye::SquarePiece;
    //...
    for(//...)
        switch(SquarePiece) // RobotEye's SquarePiece
    //...
}
```

A expressão using-directive nos permite usar as variáveis como se estas fossem definidas no escopo local.

Melhor que isto, se utilizamos constantemente diversas variáveis e funções de um dado namespace, nós podemos empregar uma expressão using-directive para criar um sinônimo local de todas as variáveis/funções deste namespace, facilitando o nosso trabalho:

```
double RobotControler::GetPiece(piece*)
{
    using namespace RobotEye; // make all names from RobotEye
                                // available in scope
    //...
    for(//...)
        switch(SquarePiece)// RobotEye's SquarePiece
    //...
}
```

19.3 Prevenindo Conflitos de Nomes

Como os namespaces foram criados para expressas estruturas lógicas, nós devemos também utilizá-los quando pretendemos distinguir diferentes partes do programa, desta forma evitaremos conflitos com os nomes declarados nestas partes.

Somebody_NeuralNetwork.h

```
namespace Somebody_NeuralNetwork
{
    //declarations and some implementation
}
```

AnotherOne_NeuralNetwork.h

```
namespace AnotherOne_NeuralNetwork
{
    //declarations and some implementation
}
```

SomeSourceCode.cpp

```
{
    // some code
    //...
    AnotherOne_NeuralNetwork::sigmoid_neuron = other_neuron;
    /* using qualification to identify variable */
    //...
}
```

Nós poderíamos usar tanto o `sigmoid_neuron` da `Somebody_NeuralNetwork's` ou da `AnotherOne_NeuralNetwork's`, mas optamos por usar a declarada em `AnotherOne_NeuralNetwork's`.

19.4 Namespaces sem Nome

Podemos também criar namespaces sem nomes, somente para criar uma relação de escopo:

```
namespace
{
    class RobotControler{//...};
    //...
}
```

esta declaração é igual a:

```
namespace $$$
{
    class RobotControler{//...};
    //...
}
```

e nós podemos acessar os seus termos da seguinte maneira:

```
using namespace $$$;
```

note que isto funciona somente para escopos locais.

19.5 Apelidos para Namespace

Nomes de namespace longos podem ser tediantes e chatos para usar:

```
namespace My_new_and_reviewed_NeuralNetwork {//...}  
//...  
My_new_and_reviewed_NeuralNetwork::sigmoid_neuron = otherneuron;
```

Nós podemos manipular isto criando apelidos para os namespaces:

```
namespace MNR_NeuralNet = My_new_and_reviewed_NeuralNetwork;  
//...  
MNR_NeuralNet::sigmoid_neuron = otherneuron;
```

19.6 Declarando Nomes

Nós não podemos nunca usar uma variável qualificada que não existe na declaração de um namespace ou tentar declarar um nome fora do escopo do namespace e tentar usá-lo no seu interior.

```
namespace RobotControler {//...}  
  
int RobotControler::new_name(arguments); //error: new_name doesn't  
                                         //exist in RobotControler
```

19.7 Atualizando Códigos Antigos

Namespaces são uma maneira fácil de migrar de códigos antigos para códigos mais atualizados sem ter que modificar demasiadamente os programas:

Old_program.h

```
void funtion(int){//...}  
//...
```

passa para:

New_program.h

```
namespace newprogram  
{  
void funtion(int){//...}  
//...  
}
```

19.8 Namespaces são Abertos

Isto significa que nós podemos adicionar nomes para este em diversas declarações, em partes distintas do programa:

Arq1.h

```
namespace NeuralNetwork
{
    Neuron* SigmoidNeuron;
}
```

Arq2.h

```
namespace NeuralNetwork
{
    Neuron* NewNeuron; //now the NeuralNetwork namespace has two
                        //members - SigmoidNeuron and NewNeuron.
}
```

20 Templates

Templates fornecem suporte direto para a programação genérica, isto é, programação onde não se conhece *a priori* o tipo empregado e utiliza-se este como parâmetro. O mecanismo de template permite que um tipo se torne um parâmetro na definição de uma classe ou função.

20.1 Classe Templates

Por exemplo, se nós queremos criar uma classe string que forneça operações básicas com caracteres, porém queremos que esta sirva para qualquer tipo de caracter, com sinal ou sem sinal, caracteres Japoneses, Chineses ou qualquer outro. Assim, nós queremos que esta classe dependa minimamente de um tipo específico de caracter. Assim sendo, devemos implementá-la de forma genérica:

```
template<class Anyone> class String
{
    struct Srep;
    Srep* rep;

    public:
    String();
    String(const Anyone*);
    String(const String&);

    Anyone read(int i) const;
    //...
};
```

o prefixo `template<class Anyone>` é usado para expressar que a classe `String` é uma classe template (genérica) e que o tipo `Anyone` pertence a sua declaração e implementação, e será definido em tempo de compilação.

20.1.1 Especificação

Agora, para a instanciação de um objeto da classe `String`, especificação, nós podemos escolher qual o tipo de caracter que nós desejamos utilizar:

```
String<char> sentence;
String<unsigned char> contents;
String<Japanese_char> cannotunderstand;
```

20.1.2 Membros da Classe Template

Membros de uma classe template são declarados e definidos exatamente como eles iguais as definições em classes normais. Se eles são implementados fora da classe template, estes precisam ser declarados explicitamente com o parâmetro da classe template:

```
template<class Anyone> struct String<Anyone>::Srep
{
    Anyone* s; // pointer to elements
    int size;
```

```
    //...
};

template<class Anyone> Anyone String<Anyone>::read(int i) const
{
    return rep->s[i];
}

template<class Anyone> String<Anyone>::String
{
    rep = new Srep;
}
```

20.1.3 ParâmetrosTemplate

Uma classe template pode receber tipos de parâmetros, tipos ordinários (int, char, long, etc.) ou também tipos template. As templates podem naturalmente receber diversos tipos de parâmetros:

```
template<class AClass, AClass def_val, String<char> ch> class Cont
{
    //...
};
```

Com visto, podemos usar argumentos template na definição de parâmetros template subsequentes e em suas implementações:

```
template<class That, int x> class Buffer
{
    That stack[x];
    int size;

    public:
    Buffer():size(x){}
    //...
};

Buffer<char,100> char_buff;
Buffer<AClass,12> aclass_buff;
```

20.2 Funções Templates

Para muitas pessoas, o mais óbvio uso de templates é na definição de classes containers como a vector, map ou a list. Logo após, surge a necessidade das funções templates, caracterizadas por manipularem tipos genéricos de dados:

```
template<class ThisOne> void sort(Vector<ThisOne>&); //declaration

void f(Vector<int>& vi, Vector<string>& vs)
{
    sort(vi); // sort(Vector<int>&)
    sort(vs); // sort(Vector<String>&)
}
```

20.2.1 Function Template Arguments

Funções templates são essenciais para escrever algoritmos genéricos. A habilidade para deduzir os argumentos template através dos parâmetros passados na chamada da função é de crucial importância. Através da chamada, é que o compilador conhece qual o tipo de dado que esta sendo empregado:

```
template<class T, int i> T& lookup(Buffer<T,i>& b, const char* p);

class Record
{
    const char[12];
    //...
};

Record& f(Buffer<Record,128>& buf, const char* pointer)
{
    return lookup(buf, pointer);
}
```

Implicitamente, o compilador pode deduzir que T é um Record e que i é 128 devido aos argumentos da função.

Algumas vezes, dedução implícita não é tão simples e, muitas vezes, faz-se necessário uma definição explícita.

```
Template<class T, class U> T implicit_cast(U u){return u;}

Void g(int i)
{
    implicit_cast(i); //error: can't deduce T
    implicit_cast<double>(i); //ok: T is double and U is int
    implicit_cast<char,double>(i); //ok: T is char and U is double
}
```

20.2.2 Sobreescrevendo Funções Templates

Funções templates também podem ser sobreescritas:

```
template<class T> T sqrt(T);
template<class T> complex<T> sqrt(complex<T>);
double sqrt(double);

void f(complex<double> z)
{
    sqrt(2);           // sqrt<int>(int)
    sqrt(2.0);         // sqrt(double)
    sqrt(z);           // sqrt<double>(complex<double>)
}
```

20.3 Especialização

Por default, uma template fornece uma definição singular para ser usada para todos os argumentos templates que o usuário possa pensar. Porém, algumas vezes, nós podemos querer tratar diferenciadamente de acordo com o tipo, como: se o argumento template é um ponteiro, faça isso, se é um double, faça aquilo, ou ainda mais, se for um tipo específico, realize tal tarefa.

```
template<class T> class Vector
{
    T* v;
    int size;

    public:
    Vector();

    T& element(int i){return v[i];}
    T& operator[](int i);
    //...
};
```

Nós poderíamos tornar isto mais específico para um vetor de ponteiros:

```
template<class T> class Vector<T*> { //... }; //specialized for pointers
```

Nós poderíamos ir além e tornar específico para ponteiros do tipo void:

```
template<> class Vector<void*> { //... }; //specialized for void pointers
```

O prefixo Template<> diz que esta é uma especialização que pode ser especificada sem o parâmetro template, por que o parâmetro específico já esta inserido nos <> após o nome da classe.

As declarações de especialização têm preferência em relação as demais.

Nós também podemos especializar funções template:

```
template<class T> bool less(T a, T b) {return a<b;} //generic

template<> bool less<const char*>(const char* a, const char* b)
{return strcmp(a,b)<0;} //for const char*

template<> bool less<>(const char* a, const char* b)
{return strcmp(a,b)<0;} //for const char*, blank brackets(redundant)
    //because of implicit recognition of types
    //from function arguments

template<> bool less<>(const char* a, const char* b)
{return strcmp(a,b)<0;} //for const char*, best declaration
```

20.4 Derivação e Templates

Templates e derivações são mecanismos para construção de novos tipos, e para escrita de códigos uteis que empregam várias formas com as mesmas funcionalidades.

Podemos derivar uma classe template da seguinte maneira:

```
template<class T> class Vector<T*>:private Vector<void*>{///  
...
```

20.5 Polimorfismo

Polimorfismo está presente em ambas classes abstratas e templates. A implementação de códigos de templates é idêntica para todos os tipos de parâmetros. Por outro lado, as classes abstratas definem uma interface comum para todas as classes derivadas.

Como ambas permitem que um algoritmo possa ser definido uma vez e aplicado para diferentes tipos, nós podemos defini-las como polimórficas. Para distingui-las, o que funções virtuais realizam é chamado polimorfismo em tempo de execução, e o que as templates realizam é chamado de polimorfismo em tempo de compilação ou polimorfismo paramétrico.

Para escolher entre as duas opções, nós precisamos analisar o seguinte:

- se nenhuma relação hierárquica entre os objetos é necessário, melhor usar templates;
- se os tipos de dados dos objetos não são conhecidos em tempo de compilação, então é melhor representá-los por classes derivadas de uma classe abstrata.

21 Container

Um container é um objeto que contém outros objetos. Exemplos são listas, vetores e arrays associativos. Em geral, você pode adicionar objetos a um container e remover objetos dele.

Estes elementos podem ser de qualquer tipo de objeto que tenha um construtor default, um destruidor e um operador de atribuição.

21.1 Iteradores

Para ter acesso ao conteúdo dos containers da mesma maneira com que se acessa arrays convencionais, é necessário o uso de iteradores. Eles nos possibilitam navegar no conteúdo dos containers sem nos preocupar com seu conteúdo. A STL inclui vários tipos de iteradores, incluindo iteradores de acesso aleatório, reversos e iteradores iostream.

21.2 Tipos de Containers

21.2.1 Containers Sequenciais

São objetos que armazenam outros objetos num arranjo estritamente linear.

- `vector<T>` : Permite o acesso aos dados como se fosse um array para uma sequência de dados de tamanho variável, com inserções e destruições no final com tempo constante;
- `deque<T>` : Permite o acesso aleatório à uma sequência de dados de tamanho variável, com inserções e destruições com tempo constante no início e no final;
- `list<T>` : Permite o acesso aos dados em tempo linear para uma sequência de dados de tamanho variável, com inserções e destruições de tempo constante em qualquer ponto da sequência.

21.2.2 Containers Associativos:

Possibilitam uma rápida recuperação de objetos da coleção baseado em chaves. O tamanho da coleção pode variar em tempo de execução. A coleção é mantida em ordem baseado em um objeto função de comparação do tipo `Compare` :

- `set<T, Compare>` : Suporta chaves únicas (contém no máximo uma chave com cada valor) e dá acesso rápido para recuperar as chaves;
- `multiset<T, Compare>` : Suporta chaves duplicadas (pode conter múltiplas cópias do mesmo valor de chave) e dá acesso rápido para recuperar as chaves;
- `map<T, Compare>` : Suporta chaves únicas (contém no máximo uma chave com cada valor) e dá acesso rápido a outro tipo T baseado nas chaves;
- `multimap<T, Compare>` : Suporta chaves duplicadas (pode conter múltiplas cópias do mesmo valor de chave) e dá acesso rápido a outro tipo T baseado nas chaves.

21.3 Exemplo de Container

No seguinte exemplo, um vetor `v` é construído e algumas operações são feitas com ele.

```
#include <iostream>
#include <vector>
using namespace std;
```

```
void pause()
{
    char lixo;
    cout << "\nPress any key to continue";
    cin >> lixo;
}

void printvector(vector<int> v)
{
    int i=0;
    cout << "Vector: ";
    for (i=0; i < v.size(); i++)
        cout << v[i] << " ";
    cout << "\n";
}

int main(int argc, char* argv[])
{
    cout << "Programa exemplo de contaneres\n\nvector<int> v(3,2)\n";
    vector< int > v(3,2);
    // Declara v como container vector do tipo int
    printvector(v);
    pause();

    cout << "\nv.push_back(5);
    // Insere o elemento 5 no final da sequencia\n";
    v.push_back(5);
    // Insere o elemento 5 no final da sequencia
    printvector(v);
    pause();

    cout << "\nInserindo mais elementos...\n";
    v.push_back(3);
    v.push_back(7);
    v.push_back(15);
    v.push_back(1);
    printvector(v);
    pause();

    cout << "\nv.pop_back(5);
    // Apaga o elemento do final da sequencia\n";
    v.pop_back();
    // Apaga o elemento do final da sequencia
    printvector(v);
    pause();
    return 0;
}
```

22 Exceptions Handling

Mesmo após o esforço de depuração da lógica e do código do software, situações excepcionais não previstas podem ocorrer durante a execução do programa e fazerem com que o programa incorra em erros ou até mesmo termine abruptamente.

Embora de tais situações, como falta de memória, entrada de dados incorretos, perda de linha do modem, disco fora da unidade, arquivo inexistente, entre outras, não possam ser evitadas pelo programador, ele deve se preparar para tratá-las da melhor maneira possível se por ventura vierem a ocorrer.

As opções existentes quando da ocorrência de algum evento como esse são:

- Derrubar o programa;
- Informar a ocorrência ao usuário e finalizar o programa;
- Informar a ocorrência ao usuário e permitir que ele tente reparar o erro e continuar a execução do programa;
- Executar uma ação corretiva automática e prosseguir sem avisar o usuário.

Derrubar o programa é o que acontece por default quando uma exceção não é chamada entretanto, para a maioria dos erros é possível fazer melhor que isto.

No caso de chamadas de funções que retornam algo, não se pode checar os valores a cada chamada, pois isso pode facilmente dobrar o tamanho do programa.

O **Tratamento de Exceções do C++** possibilita ao programador combater e tratar da melhor forma essas ocorrências, que apesar de insólitas, podem, na maioria das vezes, serem previstas durante a fase de projeto do software.

22.1 Exceção

Em C++ uma exceção é um objeto que é passado da área de código onde ocorre um erro (geralmente em um nível muito baixo do programa) para a parte do código que irá tratar o problema (normalmente um nível mais alto do programa, com código para interação com o usuário).

22.2 Como as exceções são usadas

Blocos **try** são criados para cercar áreas de código que podem conter um problema, da seguinte maneira:

```
try
{
    Função_Perigosa();
}
```

Blocos **catch** tratam as exceções aplicadas ao bloco-try:

```
try // como feito anteriormente
{
    Função_Perigosa();
}

catch(SemMemoria)
{
    //Ações a serem tomadas
```

```
}  
  
catch(ArquivoNaoEncontrado)  
{  
    //Outras ações  
}
```

Os passos básicos da utilização de exceções são:

1. Identificar aquelas áreas do programa nas quais existe uma operação que pode levantar uma exceção e inseri-las em um bloco **try**. É interessante ressaltar que essa tarefa pode ser uma das mais trabalhosas do processo de tratamento do exceções.
2. Criar tantos blocos **catch** quanto necessários para apanhar as exceções que forem aplicadas, limpar a memória alocada e passar as informações apropriadas ao usuário.

Os blocos **try** cercam as áreas do código onde podem ocorrer problemas, entretanto é preciso que as funções ou “sub funções” chamadas nessas áreas estejam aptas a requisitar a exceção caso aconteça algum evento inesperado. A exceção é aplicada utilizando a instrução **throw**, que pode passar qualquer tipo C++, inclusive objetos, para serem tratados por **catch**.

O tratador é chamado somente se o tipo passado por **throw** é o mesmo tipo aceito por **catch**.

Quando da chamada de um **catch** por um **throw**, é feita uma cópia do objeto passado para o tratador.

O exemplo de código a seguir demonstra o uso do tratamento de exceções:

```
#include <iostream>  
using namespace std;  
  
void S2iFunc( void );  
  
class S2iTest  
{  
public:  
    S2iTest(){};  
    ~S2iTest(){};  
    const char* ShowReason() const { return "Exceção na classe S2iTest!";}  
};  
  
class S2iDemo  
{  
public:  
    S2iDemo();  
    ~S2iDemo();  
};  
  
S2iDemo:: S2iDemo()  
{  
    cout << "Construindo S2iDemo." << endl;  
}  
  
S2iDemo::~ S2iDemo()  
{  
    cout << "Destruindo S2iDemo." << endl;  
}
```

```
void S2iFunc()
{
    S2iDemo D;          // cria um objeto da classe S2iDemo
    cout<< "Em S2iFunc(). Throwing a exceção S2iTest." << endl;
    throw S2iTest();    //Envia um objeto da classe S2iTest para o tratador
}

int main(int argc, char* argv[])
{
    cout << "Estamos em main." << endl;
    try
    {
        cout << "No bloco try, chamando S2iFunc()." << endl;
        S2iFunc();
    }
    catch( S2iTest E )    //Recebe um objeto da classe S2iTest
    {
        cout << "Estamos no tratador catch." << endl;
        cout << "Tratando exceção do tipo S2iTest: ";
        cout << E.ShowReason() << endl;
    }
    catch( char *str )
    {
        cout << "Trata outra exceção " << str << endl;
    }
    cout << "De volta a main. Execucao termina aqui." << endl;
    return 0;
}
```

Saída do programa:

```
Estamos em main.
No bloco try, chamando S2iFunc().
Construindo S2iDemo.
Em S2iFunc(). Throwing a exceção S2iTest.
Destruindo S2iDemo.
Estamos no tratador catch.
Tratando exceção do tipo S2iTest:
Exceção na classe S2iTest !
De volta a main. Execucao termina aqui.
```

22.3 Biblioteca Except <except.h>

Esta biblioteca dá suporte para exceções em ANSI C++. Algumas funções dessa classe são descritas a seguir.

22.3.1 Terminate()

```
void terminate();
```

A função *terminate()* pode ser chamada por outra função (como *unexpected*) ou pelo programa quando um tratador para uma exceção não for encontrado. A ação default para *terminate* é chamar *abort*, causando imediatamente o término do programa.

O modo de término pode ser modificado utilizando-se de *set_terminate*.

terminate() não retorna valor pois é uma função *void*.

22.3.2 Set_terminate()

```
typedef void (*terminate_function)();  
terminate_function set_terminate(terminate_function t_func);
```

Permite criar uma função que define o comportamento do término do programa se um tratador de exceção não for encontrado.

As ações a serem realizadas em tal situação são definidas em *t_func*, que é declarada como uma função do tipo *terminate_function*. O tipo *terminate_function* é definido em `<except.h>` e é uma função que não recebe argumentos e retorna *void*.

Se nenhum tratamento de exceção for encontrado, por *default* o programa chama a função *terminate*, resultando normalmente numa chamada para *abort*. Se o programador desejar uma outra ação, deve defini-la na função *t_fun*, que será instalada por *set_terminate* como a função de termino.

22.3.3 Unexpected()

```
void unexpected();
```

Se uma função envia uma exceção que não esta definida o programa chama *unexpected*. Se nenhuma ação especifica é e definida por *set_unexpected*, a função *unexpected* chama *terminate*.

22.3.4 Set_unexpected()

```
typedef void ( * unexpected_function )();  
unexpected_function set_unexpected(unexpected_function unexpected_func)
```

Essa função permite criar uma função que define o comportamento do programa se uma função envia (*throw*) uma exceção não especificada. As ações são definidas em *unexpected_function*.

O tipo *unexpected_function* está definido em `<except.h>` e não recebe argumentos e retorna *void*.

Trabalho 1

Crie um programa que desenhe na tela o seu quadro de horários deste semestre.

Trabalho 2

A decomposição LDU é uma decomposição famosa de matrizes aplicada para calcular inversas de matrizes e resolver problemas de equações lineares. Utilize o programa 9.1 e crie um programa que calcule a decomposição LDU de uma matriz, preferencialmente, com pivotamento.

Trabalho 3

Crie um programa que faça o controle de cadastro de usuários e controle de alocação para uma locadora de fitas de vídeo. Utilize uma classe para os clientes e outra para as fitas, e empregue o método da lista ligada para gerenciar um vetor dinâmico. O usuário de poder incluir/retirar um cliente ou uma fita, fazer uma pesquisa, pedir a lista de clientes e a lista de fitas ou alterar algum parâmetro destas.

Trabalho 4

Este problema foi retirado do livro “*Fundamentos da Automação Industrial Pneumática*” de Arno Bollmann (Professor da UFSC, Ed. ABHP – São Paulo). O objetivo aqui é utilizar uma ferramenta orientada a objetos para modelar, analisar e controlar um sistema pneumático automatizado. Enfocando a aplicação desta ferramenta na análise e desenvolvimento de sistemas automatizados. A ferramenta aplicada será a programação orientada a objetos em C++ juntamente com a técnica de modelagem UML.

Manipulação de peças: alimentação de uma forja a partir de três fornos.

A figura a seguir (Figura 5) apresenta o esquema de um sistema de manipulação de peças, cuja tarefa é a de alimentar uma forja com peças incandescentes apanhadas de três fornos, na medida em que os mesmos disponibilizem a peça na temperatura ajustada, sem ordem seqüencial predeterminada.

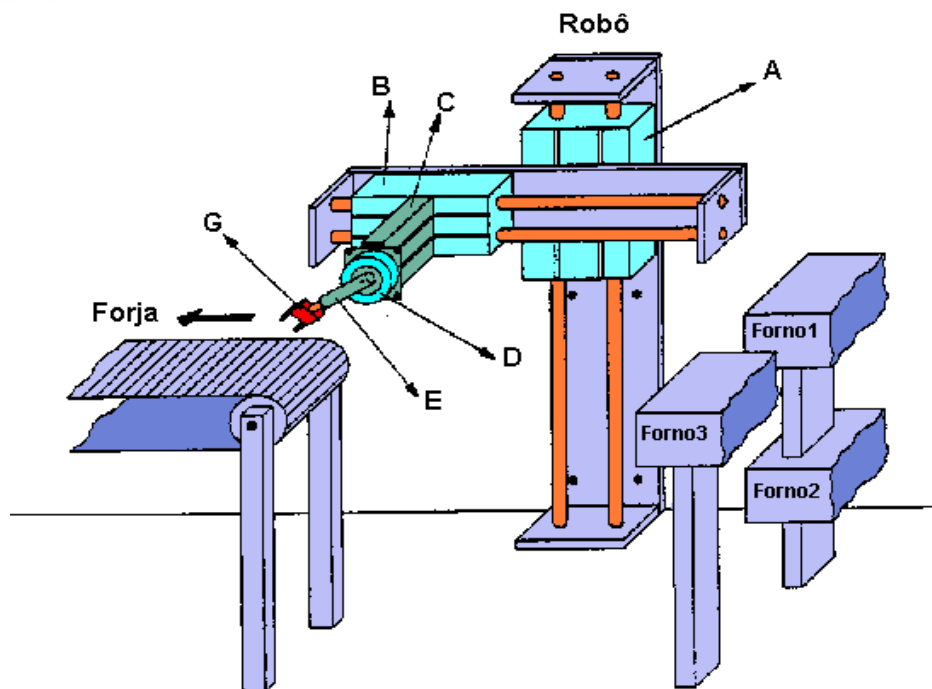


Figura 5 : O problema de manipulação de peças.

Os fornos F1 e F2 e a entrada da Forja se situam num mesmo plano vertical, traseiro, enquanto que o forno F3 está mais à frente, num plano alcançado pelo avanço do atuador C. A forja e os fornos F1 e F3 estão numa posição mais alta, num mesmo plano horizontal, correspondendo à posição do atuador A recuado, como indicado na Figura 5.

As peças podem ser colocadas aleatoriamente nos três fornos, com tempos de aquecimento não necessariamente iguais, fazendo com que as peças fiquem prontas também em ordem aleatória. Assim que a peça está pronta, o forno abre a sua porta automaticamente. O sinal de abertura da porta do forno (F1, F2 ou F3) é utilizado como o sinal de início do trabalho do manipulador. O projeto do comando seqüencial do manipulador leva em conta que esse sinal permanece até o momento em que a peça é retirada do forno. O detalhamento do funcionamento dos fornos, da sua alimentação e dos comandos de abertura e fechamento das suas portas não faz parte da solução deste exemplo. Em suma: assim que algum forno der o sinal de peça pronta, disponível, o manipulador deverá ir buscá-la e posicioná-la na forja. Após depositar a peça na forja, o manipulador aguarda em sua posição de repouso, até que fique pronta uma nova peça em qualquer um dos três fornos. Conforme mostra a Figura 5, essa posição de repouso corresponde a uma posição de recuo de todos os atuadores. Todos eles são comandados por válvulas de simples solenóide, com exceção do atuador giratório D, cuja válvula possui duplo solenóide.

A tabela a seguir lista os atuadores pneumáticos que compõem o manipulador, descreve sua função, apresenta a notação e a correspondência lógica dos seus fins de curso e indica os solenóides das válvulas de comando correspondentes, além de auxiliar a compreensão do funcionamento do manipulador.

Atuador Função do Manipulador

- A** Posiciona a garra no plano superior, para os fornos F1 e F3 e a forja, quando recuado, ou no plano inferior para alcançar o forno F2, Quando avançado.
 - B** Posiciona a garra à esquerda junto à forja, quando recuado. Avançado, alcança os fornos F1, F2 e F3.
 - C** Avançado, posiciona a garra no plano anterior, referente ao forno F3. Recuado, situa a garra no plano posterior, dos fornos F1 e F2 e da forja.
 - D** Atuador giratório, alimentado por válvula de duplo solenóide. Acionada, faz um giro de 135° no sentido anti-horário para colocar o cilindro E na posição horizontal quando a garra é levada em direção dos fornos. Recuada, esta retorna a posição original, utilizada para colocar a peça na forja.
 - E** Quando avança, introduz a garra nos fornos ou na esteira da forja. Deve sempre ser recuada antes do cilindro D realizar o giro.
 - G** Garra comandada por simples solenóide. Quando acionada, segura a peça.
-

Condições adicionais:

O comando deve ainda atender às seguintes exigências:

- ✓ além da modalidade de funcionamento automático (ciclo contínuo), deve-se ainda prever a possibilidade do comando manual passo a passo, para ajuste de cada um dos passos;
- ✓ se mais de um forno estiver simultaneamente aberto com peça disponível para ser apanhada, deve prevalecer a hierarquia de F1 antes de F2 ou F3 e a prevalência de F2 sobre F3.

Para resolver este problema, utilize uma representação detalhada do robô para poder controlar e analisar os seus movimentos. Em seguida, utilize uma representação simplificada, para analisar e controlar a interação do robô com os demais mecanismos da célula.

Pede-se:

- i. Modelagem de cada elemento do robô;
- ii. Identificação e modelagem das especificações do robô, ou seja, implementar o comportamento do robô que atenda as especificações;
- iii. Modelagem dos demais elementos da célula de produção: fornos, esteira, forja, peças; respeitando os diferentes tipos;
- iv. Modelagem da célula flexível de manufatura inteira e implementação de um controle que atenda as especificações;

Como o programa deve funcionar:

O usuário utilizará um conjunto de instruções para comandar a célula flexível de manufatura, como se estivessem operando no prompt do DOS. Sempre que a célula tiver completado uma tarefa, ela deve perguntar ao usuário qual a próxima tarefa a ser realizada. Permita que o usuário acompanhe passo a passo a evolução do sistema, apresentando na tela como o robô está se movimentando. Por exemplo, indique qual a posição atual do robô. Você não precisa criar uma animação (desenho) para o robô, basta indicar a posição dele na tela.

O conjunto de comandos a ser implementado é:

0 : nenhuma peça pronta nos fornos, robô deve se dirigir para o estado inicial.
1 : peça pronta no forno 1 (ir para o forno 1)
2 : peça pronta no forno 2 (ir para o forno 2)
3 : peça pronta no forno 1 e 2
4 : peça pronta no forno 3
5 : peça pronta no forno 1 e 3
6 : peça pronta no forno 2 e 3
... : (utilize a combinação 0, 1, 2 e 4 para saber quando tem peças prontas e em que fornos.)
o : levar a (s) peça (s) para a esteira da forja.
x : terminar o programa e escrever na tela quantas peças de cada tipo foram fabricadas.

Como você deve proceder para a realização deste trabalho:

Utilize os Anexos 1 e 2, as referências bibliográficas e os hot-sites para desenvolver o seu software segundo os seguintes passos:

1. Apresentação dos Requisitos do Programa. Descreva aquilo que lhe é pedido.
2. Análise dos Requisitos. Discussão dos requisitos fornecidos e como estem devem ser atendidos. Planejamento macro da estrutura do programa. Definição de alguns desenhos esquemáticos que ajudem no entendimento do problema e na análise dos requisitos do programa.
3. Modelagem Orientada a Objetos: construção de um modelo de classes orientado a objetos que representa o problema e atende as suas especificações. Utilize a metodologia UML.
4. Implementação do Programa: Implemente o programa em C++ que atenda as especificações feitas.
5. Testes: Defina um conjunto de testes e os realize, para verificar se o programa atende satisfatoriamente as necessidades do usuário.

Este é um desafio para aqueles que querem efetivamente aprender a programar e criar sistemas orientados a objetos e não serem meros digitadores de códigos em C. Trabalhe com profissionalismos e dedicação, faça deste trabalho o seu produto. Dedique a ele o respeito e o esforço que a tua profissão merece.

Referências Bibliográficas

- [1] Booch, G.; Jacobson, I. e Rumbaugh, J.: ***“The Unified Modelling Language – User Guide”***, Ed. Addison Wesley-Logman, 1999.
- [2] Booch, G.; Jacobson, I. e Rumbaugh, J.: ***“The Unified Software Development Process”***, ed. Addison Wesley Longman, January of 1999.
- [3] Borland C++: ***“Programmer’s Guide”***, Borland International, USA, 1994.
- [4] Borland C++: ***“User’s Guide”***, Borland International, USA, 1994.
- [5] Chapman, D.: ***“Teach Yourself Visual C++ 6 in 21 Days”***, SamsPublishing, 1998.
- [6] Douglass, B. P.: ***“Real-Time UML : Developing Efficient Objects for Embedded Systems”***, ed. Addison Wesley Longman, August of 1998.
- [7] Meyer, Bertrand .: ***“Object-Oriented Software Construction”***. Ed. Prentice Hall.
- [8] Microsoft: ***“MSDN - Microsoft® Visual Studio™ 6.0 Development System”*** Microsoft®, 1999.
- [9] Mizrahi, Victorine Viviane.: ***“Treinamento em Linguagem C – Curso Completo”***, Módulo 1, Ed. Makron Books do Brasil, São Paulo,1990.
- [10] Mizrahi, Victorine Viviane.: ***“Treinamento em Linguagem C++ - Módulo 1 e 2”***, Ed. Makron Books do Brasil, São Paulo, 1994.
- [11] Montenegro, Fernando; Pacheco, Roberto: ***“Orientação a Objetos em C++”***, Ed. Ciência Moderna, Rio de Janeiro, 1994.
- [12] Petzold, Charles: ***“Programming Windows 95”***, Microsoft Press, Washington, 1996.
- [13] Reidorph, Kent: ***“Teach Yourself Borland C++ Builder 3 in 21 Days”***, Sams Publishing, Indianapolis, USA, 1998.
- [14] Rumbaugh, J.; Blaha, M.; Premerlan, W.: Eddy, F. e Lorensen, W.: ***“Object-Oriented Modelling and Design”*** . Ed. Prentice Hall.
- [15] Stevens, W. R.: ***”Programmieren von Unix-Netzen – Grundlagen, Programmierung, Anwendung“***, Ed. Hanser and Ed. Prentice-Hall, 1992.
- [16] Stroustrup, B.: ***“C++ Programming Language”*** 3rd Edition, Addison Wesley, 1997.
- [17] Swan, Tom: ***“Programação Avançada em Borland C++ 4 Para Windows”***, Ed. Berkeley Brasil, São Paulo, 1994.

Hot Sites

- <http://www.developer.com/>
- <http://www.borland.com/>
- <http://msdn.microsoft.com/resources/devcenters.asp>
- <http://search.microsoft.com/us/dev/>
- <http://www.rational.com/uml/>
- <http://www.togethersofter.com/together/togetherC.html>
- <http://world.isg.de/world/>
- <http://www.popkin.com/services/enterprise/uml/uml.htm>
- <http://www.ee.cooper.edu/resource/docs/linux/LDP/lpg/node1.html>
- <http://www.geog.leeds.ac.uk/staff/i.turton/unix/course.html>
- <http://www.geog.leeds.ac.uk/staff/i.turton/unix/course.html>
- <http://www.codeguru.com/>
- <http://sf.net/>
- <http://www.bloodshed.net/>
- <http://www.sun.com/staroffice/>
- <http://www.cvshome.org/>
- <http://www.proxysource.com/>
- <http://www.doxygen.org/download.html>
- <http://members.tripod.com/~SoftechSoftware/index.html>
- <http://www.sgi.com/tech/stl/index.html>
- <http://www.wxwindows.org/>
- <http://argouml.tigris.org>
- <http://www.gentleware.com/products/download.php3>
- <http://www.planetpdf.com/mainpage.asp?webpageid=315>
- <http://doc.lcmi.ufsc.br/bookshelf/>
- <http://www.computer.org/publications/dlib/>



Anexo 1 – Metodologia para o Desenvolvimento de Softwares

Anexo 2 – Sumário da Modelagem UML

This document was created with Win2PDF available at <http://www.daneprairie.com>.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.