CURSO DE C++

Alceu Heinke Frigeri
Bernardo Copstein
Carlos Eduardo Pereira

Porto Alegre - RS - Janeiro de 1996

1. Introdução à programação orientada a objetos

Os primeiros sistemas de computadores eram programados por chaves no painel. A medida em que foram crescendo os programas, surgiram os montadores assembler onde o programador podia se valer de mnemônicos para facilitar o trabalho de programação. Como os programas continuaram crescendo, surgiram as linguagens de alto nível que ofereceram melhores ferramentas para o programador enfrentar a crescente complexidade dos programas. Fortran certamente foi a primeira linguagem de alto nível de grande utilização. Apesar de ser um primeiro passo, não se pode dizer que é uma linguagem que estimula a construção de programas organizados.

Nos anos sessenta surgiu a programação estruturada, base de linguagens como "C" e Pascal. Utilizando-se a programação estruturada pôde-se, pela primeira vez, escrever com bastante flexibilidade programas de complexidade moderada. Se o sistema cresce demais, no entanto, os recursos da programação estruturada passam a ser insuficientes. Os maiores efeitos deste fato podem ser sentidos quando da manutenção dos programas. Por exemplo, imagine o conjunto de funções e procedimentos que compõem um programa de bom tamanho alterando diretamente ou através de parâmetros as diferentes estruturas de dados de um sistema. A alteração de um destes procedimentos pode ter "efeitos colaterais" imprevisíveis e de alcance difícil de ser determinado.

A programação orientada a objetos conservou todas as boas idéias da programação estruturada e acrescentou alguns aspectos novos que permitem uma nova abordagem de programação. Uma das idéias básicas em programação orientada a objetos diz que o projeto do sistema deve ser feito a partir das entidades do mundo real e não a partir dos dados ou procedimentos. A maior justificativa para tanto é que dados e procedimentos mudam com muita facilidade enquanto que as entidades do mundo real tendem a ser mais estáveis.

Na modelagem de objetos do mundo real, no entanto, percebe-se que existem conjuntos de objetos com características comuns. "Classifica-se" este grupo, então, criando-se uma classificação ou "classe" para o mesmo. Em uma linguagem orientada a objetos, uma classe corresponderá a um tipo abstrato de dados, ou seja, um conjunto de atributos que descrevem a classe e o os procedimentos que podem agir sobre esses atributos. Ao conjunto de atributos e procedimentos daremos o nome de propriedades da classe. Os diferentes objetos do mundo real são então representados por instâncias dessas classes. Em um sistema de controle de um estacionamento, por exemplo, uma classe possível seria a classe veículo. Poderíamos ter como atributos de veículo o modelo, a placa, a hora de chegada e a hora de saída. Associados a esses atributos poderíamos ter métodos de consulta que nos informassem por exemplo a placa do veículo ou a quanto tempo o mesmo está estacionado. Um determinado veículo é representado como uma instância desta classe (figura 2).

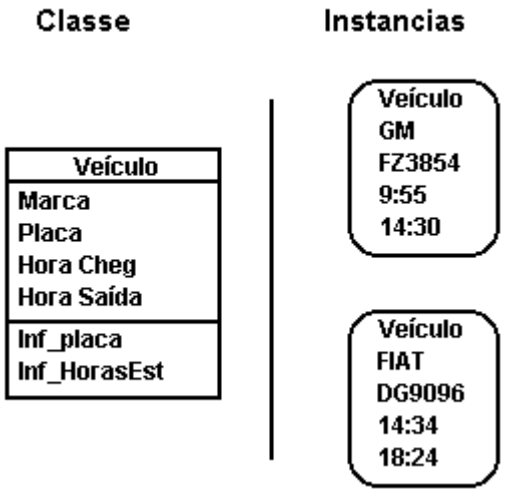


Fig. 1 - Classes x Instâncias

Uma das vantagens de se trabalhar dessa maneira é o encapsulamento dos atributos e procedimentos. Os atributos e os procedimentos que atuam sobre os mesmos se encontram agrupados permitindo que se tenha uma melhor definição do escopo dos procedimentos e uma maior segurança no acesso aos dados. É diferente de uma linguagem de programação convencional onde os dados são declarados de maneira independente das funções. Nestas, a responsabilidade de manter a conexão entre os dados e as funções cabe exclusivamente ao programador. Outra possibilidade, é a definição em separado da interface dos métodos de sua implementação.

Desta forma se alteramos a implementação de um método (por questões de otimização ou correção de erros) não haverão repercussões além do escopo da classe, ou seja: se a interface é mantida, o método continua respondendo da mesma maneira. Mesmo a alteração de um atributo pode ser feita seguindo-se este princípio. No exemplo anterior a alteração do atributo placa de maneira a permitir placas com 3 letras ao invés de duas, não terá maiores consequências. Provavelmente as alterações irão se restringir ao método "inf_placa" através do qual temos acesso a mesma.

A idéia de definição de interface permite ainda definir atributos privados, aos quais só se tem acesso através de procedimentos de consulta. A possibilidade de se definir atributos e procedimentos privados para uma classe oferece pelo menos duas vantagens: a) permite que se oculte aspectos da implementação que não interessam ao "usuário" da classe; b) facilita a manutenção da integridade dos dados uma vez que o acesso pode ser limitado a procedimentos que garantem essa integridade.

O paradigma de programação orientada a objetos diz que todo o objeto deve ter um identificador único. Este princípio é baseado na idéia de que todos os objetos do mundo real tem identidade própria. Desta forma, um objeto deverá ter uma identidade própria que o diferencie dos demais mesmo quando todos os atributos que descrevem dois objetos de uma mesma classe tiverem conteúdo idêntico. Em "C++", em geral, esse identificador único será representado por uma referência ao objeto (endereço). Esse tipo de identidade, entretanto, terá de ser substituído caso seja necessário armazenar os objetos em memória secundária.

Outras dois conceitos importantes das linguagens orientadas a objetos são subclasse e polimorfismo.

O conceito de subclasse permite criar classes derivadas de classes mais genéricas com o objetivo de detalhar melhor um subconjunto dos componentes da superclasse. A principal característica é que a subclasse herda todos os atributos e procedimentos da superclasse e acrescenta os que lhe são particulares. No exemplo 3, as subclasses "func gerente" e "func superv" herdam os atributos e os procedimentos da superclasse funcionário, além de acrescentar aqueles que as particularizam. Com esta estrutura é possível criar instâncias de funcionários comuns, funcionários gerentes e funcionários supervisores. Observe que cada tipo de funcionário tem seu próprio método para cálculo de salário, embora todos compartilhem o método de cálculo do valor hora (re-utilização de código).

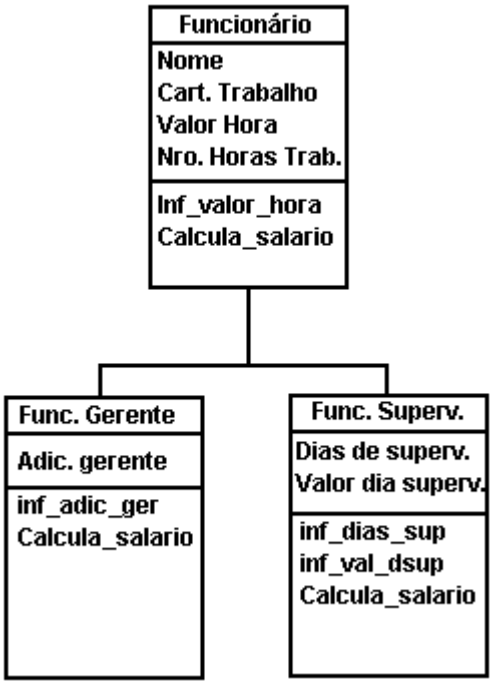


Fig. 2 - Herança

O conceito de polimorfismo permite que se especialize um método da superclasse em algumas ou todas as subclasses. No exemplo da figura 3, o método "calcula salário" é especializado em cada uma das subclasses pois o cálculo do salário apresenta diferenças para cada categoria. No momento da ativação do método "calcula salário", conforme a classe ou subclasse que o objeto pertença, o procedimento correto será ativado sem a necessidade de testes como em uma linguagem de programação comum.

Finalmente podemos modelar nossos objetos utilizando o conceito de agregação. Em uma agregação procura-se representar relacionamentos do tipo "é parte de" e não do tipo "subclasse de" como no exemplo anterior. Um objeto *carro* pode ser formado por um objeto *porta* e um objeto *motor* entre outros. *Motor* e *porta* são

objetos que podem existir por si mesmos enquanto que um "Gerente" é antes de tudo um "Funcionário". Representações desse tipos são obtidas quando os objetos possuem atributos que são instâncias de objetos.

2. Histórico - As origens do "C++"

Visando contornar os problemas de manutenção existentes nos grandes sistemas, em 1980 Bjarne Stroustrup adicionou o conceito de classes e de verificação de parâmetros de funções além de algumas outras facilidades à linguagem "C". A linguagem resultante foi chamada de "C com classes". Em 1983/84 o "C com classes" foi extendido e re-implementado resultando na linguagem conhecida como "C++". As maiores extensões foram as funções virtuais e a sobrecarga de operadores. Após mais alguns refinamentos a linguagem "C++" tornou-se disponível ao público em 1985 e foi documentada no livro "The C++ Programming Language" (Addison Wesley 1986).

2.1 Generalidades

A linguagem "C++" pode ser considerada como uma extensão da linguagem "C". Os primeiros compiladores "C++" não eram 'nativos', uma vez que não geravam diretamente código executável, mas convertiam código "C++" em código "C" para uma posterior compilação com um compilador "C" comum. A figura 1 mostra o relacionamento entre as duas linguagens.

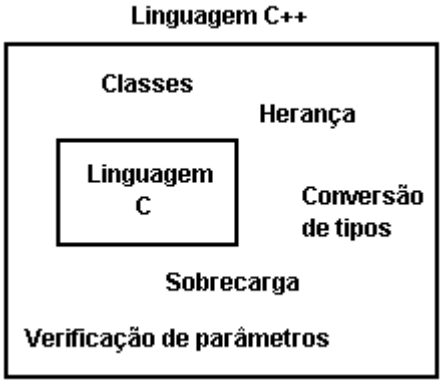


Fig. 3 - Relação entre a linguagem "C" e a linguagem "C++"

A linguagem "C++" acrescenta à linguagem "C" facilidades que permitem o desenvolvimento de uma programação orientada a objetos sem, entretanto, desfazer-se das características da linguagem "C". Se por um lado ganha-se uma ferramenta extremamente poderosa deve-se tomar alguns cuidados. A linguagem "C++" permite operações que normalmente só são aceitas em linguagens de baixo nível ao mesmo tempo em que oferece facilidades para programação orientada a objetos. Isso implica em que a linguagem oferece as condições mas não impõem a disciplina, exigindo portanto programadores experientes e disciplinados. Assim como na linguagem "C", todas as liberdades oferecidas exigem responsabilidade por parte do programador.

3. Compatibilidade entre programas "C' e "C++"

Como já foi enfatizado anteriormente, "C++" é uma extensão da linguagem "C". Por ser uma extensão, é natural supor que um programa "C" seja compilado sem problemas com um compilador "C++", já que este compilador deve ser capaz de compilar todos os comandos da linguagem "C" mais os comandos acrescentados que compõem o "C++". Neste capítulo são apresentadas algumas diferenças de estrutura entre as linguagens "C" e "C++" que não dizem respeito aos comandos que suportam programação orientada a objetos e que permitirão verificar quando um programa "C" pode ser compilado em um programa "C++" e vice-versa.

a) Prototipação das funções: a prototipação das funções em "C++" é obrigatória. Nenhuma função pode ser utilizada dentro de um módulo de programa sem anteriormente ter sido totalmente prototipada ou declarada, isto é, o número e tipo dos parâmetros das funções devem ser conhecidos (isto revela que a 'tipagem' é mais forte em C++ que em C.

b) Declaração de variáveis: embora a declaração de variáveis ocorra do mesmo modo tanto em "C" como em "C++", em "C++" a ordem de declaração é mais flexível. Em "C" todas as declarações devem ocorrer no início de um bloco de função ou bloco de comandos. As variáveis devem ser declaradas não apenas antes de serem executadas, mas antes de qualquer proposição executável. Em "C++" as declarações podem ser colocadas em qualquer ponto do programa. Na verdade devem ser colocadas o mais próximo possível do ponto de utilização de forma a salientar ao programador sua relação com as proposições que a utilizam e lembrar ao programador sobre seu escopo.

c) Escopo das variáveis: as regras de escopo são semelhantes em "C" e "C++". Existem 2 tipos comuns de escopo: local e file. "C++" apresenta ainda o tipo de escopo **class** que será discutido mais adiante. Em "C", no caso de ocorrerem identificadores locais idênticos a identificadores globais, o identificador global fica "mascarado" pelo identificador global. Em "C++" a regra é a mesma, porém essa regra pode ser quebrada com o uso do operador de resolução de abrangência (escopo): "**::**".

Exemplo:

```
int cout = 5;
int exemplo()
  {
    int count = 10;
    printf("Contagem externa: %d\n",::count);
    printf("Contagem interna %d\n",count);
  }
```

d) Comentários: os comentários em "C++" podem ser de dois tipos:

1) Bloco de comentário: delimitado por um par: ***/**** comentario ****/***

2) Comentário de uma linha: é delimitado por um "***//***" e pelo final de linha

Ex:

```
// Comentario de uma linha
```

e) Casts: os "casts" podem ser usados em "C++" da mesma forma que em "C", porém o "C++" oferece uma forma alternativa que enfatiza melhor o escopo de abrangência do "cast". São expressões válidas em "C++":

```
x = (float)a + b / 2.0;
x = float(a) + b / 2.0; // ver construtores adiante
```

f) Argumentos de funções: em "C++" os argumentos de funções devem ser declarados conforme o padrão ANSI. O estilo K&R não é aceito.

Exercícios:

1) Analise o fonte "exer1.c". Observe as declarações de variáveis e prototipação exclusivas do "C++". Compile o programa com um compilador "C" e observe o tipo de mensagem de erro. Compile, depois, com um compilador "C++" e execute o programa.

2) Altere o uso do "cast" no programa "exer1.c" para o estilo do "C++".

3) Comente a linha que contém o protótipo da função "media" e observe o tipo de mensagem de erro (use o comentário de linha do "C++").

4) Acrescente uma variável global chamada "size" ao programa "exer1.c" e inicialize-a com o valor 5. Altere em seguida a função "media" para utilizar o valor desta variável ao invés do parâmetro "size" com auxílio do operador de resolução de abrangência.

4. Características únicas das funções em C++

As funções da linguagem "C++" possuem uma série de aprimoramentos concebidos para tornar mais fácil sua programação e uso.

4.1 Funções Inline

Quando uma determinada função é muito pequena ou simples, o custo de chamada da função (empilhamento dos parâmetros e do endereço de retorno, desvio etc), pode ser proibitivo principalmente se a função for ativada muitas vezes. Uma prática bastante conhecida é o uso de "funções macro" construídas com o auxílio de pré-processador. Não sendo funções, mas sim expansões de macro, não possuem o custo de uma função e evitam a desestruturação do programa.

Exemplo:

```
#define maior(a,b)   (a > b) ? a : b
void main()
  {
     int x,y,m;
     x=4; y= 3;
     m = maior(x,y);
     printf("%d\n",m);
  }
```

O grande problema associado ao uso de macros é que a verificação de erros normalmente feita para uma função não ocorre.

Para contornar esse problema o "C++" introduziu as funções "**inline**". Uma função definida como "**inline**" não gera código. Toda vez que a função é chamada, seu código é expandido no ponto de chamada como se fosse uma macro.

Exemplo:

```
inline int soma(int a,int b)
  {
     return(a+b);
  }
```

Funções "**inline**" devem ser usadas com cuidado. Embora gerem um código mais rápido, o tamanho do mesmo pode crescer de maneira assustadora na proporção em que a função é ativada muitas vezes. A declaração "**inline**" foi criada com o propósito de ser utilizada com funções pequenas. Um caso típico são as funções de consulta a um objeto. Outra grande vantagem do uso de funções "**inline**" é que uma função "**inline**" definida, porém, não utilizada, não irá gerar código nenhum.

4.2 Sobrecarga de funções

A linguagem "C++" apresenta o conceito de sobrecarga de função. Isso significa que uma função pode ser chamada de vários modos, dependendo da necessidade. Em geral usa-se sobrecarga entre os métodos de uma classe, mas pode se usar com qualquer função.

Exemplo:

```
int soma(int a,int b)
  {
     return(a+b);
  }
double soma(double a,double b)
  {
     return(a+b);
  }
int soma(int a,int b,int c)
  {
     return(a+b+c);
  }
```

```
void main()
  {
     int s1,s2;
     double s3;
     s1 = soma(23,89);
     s2 = soma(1,4,8);
     s3 = soma(4.57,90.87);
     printf("%d %d %f\n",s1,s2,s3);
  }
```

As funções podem ser sobrecarregadas quanto ao tipo dos parâmetros, quanto a quantidade dos mesmos ou ambos. Para manter as normas de programação orientada a objeto, entretanto, o significado semântico da função não deve mudar de uma implementação sobrecarregada para outra. Em resumo, todas as versões da função devem fazer a mesma coisa. Importante: é responsabilidade do programador garantir esta consistência, uma vez que o compilador não faz nenhuma checagem quanto à semântica da operação.

4.3 Argumentos padrão (‘default’)

Uma função em "C++" pode ter argumentos com valores pré-definidos que o compilador usa quando outros novos valores não são fornecidos. Os argumentos default são especificados quando a função é declarada. Uma função pode ter vários argumentos com valores default, porém, eles devem ser os últimos argumentos da função.

Exemplo:

```
void teste(int a,int b,int op=1)
  {
     int aux;
     aux = a*b;
     if (op)
       if (aux < 0) aux *= -1;
     printf("%d\n",aux);
  }
void main()
  {
     teste(-10,20);
     teste(-10,20,0);
  }
```

Argumentos padrão são úteis quando um determinado valor para um argumento se repete na maior parte das chamadas ou em funções com grande número de argumentos que não precisam ser necessariamente especificados (funções de janela por exemplo).

Exercícios:

1) Analise o fonte "exer2.cpp". Substitua as funções "aski" e "askf" por duas versões sobrecarregadas de uma função chamada "ASK".
2) Altere as funções "ASK" e "impperg" fazendo com que o parâmetro "nl" passe a ter o valor padrão "0". Altere, também, as chamadas correspondentes para que informem o valor para este parâmetro apenas quando for necessário.
3) Analise o fonte "exer3.cpp". Compile-o e verifique seu tamanho. Retire as declarações "**inline**" e observe as alterações no tamanho do código.
4) Experimente acrescentar na função **inline** do "exer3.cpp" um comando "**for**" ou "**switch**" e observe o tipo de mensagem de erro.
5) Porque a função "teste" do "exer3.cpp" não precisa de protótipo?

5. Introdução a Classes e Objetos

5.1 Introdução

Vamos analisar inicialmente o seguinte conjunto de rotinas e tipos de dados que implementam uma estrutura de dados do tipo pilha:

```
#include <stdio.h>
#include <stdlib.h>
#define TAM_MAX_PILHA 100
typedef struct
  {
    int pilha[TAM_MAX_PILHA];
    int tamanho;
    int topo;
  } TPILHA;
void init_pilha(TPILHA *p,int tam)
  {
    int r;
      /* Verifica o tamanho */
    if (tam >= TAM_MAX_PILHA) exit(0);
    else p->tamanho = tam;
      /* Zera o vetor que serve de base a pilha */
    for(r=0; r<p->tamanho; r++)
      p->pilha[r] = 0;
      /* Inicializa o topo */
    p->topo = 0;
  }
void ins_pilha(TPILHA *p,int num)
  {
      /* Verifica se ha espaco na pilha */
    if (p->topo == p->tamanho) exit(0);
      /* Acrescenta o valor na pilha */
    p->pilha[p->topo++] = num;
  }
int ret_pilha(TPILHA *p)
  {
      /* Verifica se a pilha nao esta vazia */
    if (p->topo == 0) exit(0);
      /*Retorna o elemento do topo */
    return(p->pilha[p->topo--]);
  }
void main()
  {
    TPILHA p1;
    int r;
    init_pilha(&p1,50);
    for(r=0; r<50; r++)
      ins_pilha(&p1,r);
    for(r=0; r<50; r++)
      printf("%d\n",ret_pilha(&p1));
  }
```

O programa acima, apresenta uma estrutura de dados do tipo pilha implementada através de uma estrutura de dados do tipo "**struct**" (TPILHA) e de três rotinas: init_pilha, ins_pilha e ret_pilha. Se esses elementos forem usados dentro de certas regras pode-se com facilidade simular a estrutura de dados pilha. Entre essas regras pode-se destacar:

- O acesso ao vetor "pilha" deve ser feito sempre através das rotinas;
- O usuário não deve esquecer jamais de usar a rotina "init_pilha" antes de poder começar a trabalhar com uma variável do tipo TPILHA como se fosse uma pilha.

- Uma referência para a variável que contém a pilha deve ser sempre passada como parâmetro para todas as funções que agem sobre a pilha.

As regras acima são necessárias porque não existe uma vinculação no programa entre a estrutura de dados TPILHA e as rotinas apresentadas. Essa vinculação é responsabilidade do programador, cabendo a ele o uso correto das ferramentas oferecidas.

Uma das principais características da linguagem "C++" é o conceito de objetos e de sua estrutura, a classe. Uma classe define um tipo de dados abstrato que agrupa tanto campos de dados (da mesma forma que uma estrutura) como rotinas.

Uma classe é como uma declaração de tipo de dados. Assim como todas as variáveis de um determinado tipo, todos os objetos de uma dada classe possuem seus próprios campos de dados (atributos) e rotinas (procedimentos), mas compartilham os nomes desses atributos e procedimentos com os demais membros da mesma classe.

Sendo assim poderíamos usar uma declaração de classe para agrupar as rotinas e os campos de dados do exemplo anterior. Teríamos, então, a classe PILHA.

Ex:

```
#define TAM_MAX_PILHA 100
class PILHA
  {
    int pilha[TAM_MAX_PILHA];
    int tamanho;
    int topo;
    public:
    void init(int tam);
    void ins(int num);
    int  ret(void);
  };
```

Uma classe pode conter tanto elementos públicos quanto privados. Por definição, todos os ítens definidos na "**class**" são privados. Desta forma os atributos "pilha", "tamanho" e "topo" são privados. Desta forma não podem ser acessados por nenhuma função que não seja membro da "**class**". Essa é uma maneira de se obter o encapsulamento, ou seja, o acesso a certos itens de dados pode ser controlado.

As partes públicas da classe, ou seja, aquelas que podem ser acessadas por rotinas que não são membros da classe devem ser explicitamente definidas depois da palavra reservada "**public**". Pode-se ter tanto atributos como procedimentos na parte pública de uma classe embora uma boa norma de programação seja restringir os atributos de dados a parte privada.

Uma vez definida uma classe pode-se criar objetos daquele tipo.

Ex:

```
PILHA p1;
```

Dentro da declaração da classe forma indicados apenas os protótipos das funções membro da classe. Quando chega o momento de codificar uma das funções membros, deve-se indicar a qual classe a mesma pertence. A declaração completa da classe fila fica então:

```
#define TAM_MAX_PILHA 100
class PILHA
  {
    int pilha[TAM_MAX_PILHA];
    int tamanho;
    int topo;
    public:
    void init(int tam);
    void ins(int num);
    int  ret(void);
  };
void PILHA::init(int tam)
  {
    int r;
```

```
      /* Verifica o tamanho */
    if (tam >= TAM_MAX_PILHA) exit(0);
    else tamanho = tam;
    /* Zera o vetor que serve de base a pilha */
    for(r=0; r<tamanho; r++)
      pilha[r] = 0;
    /* Inicializa o topo */
    topo = 0;
  }
void PILHA::ins(int num)
  {
    /* Verifica se ha espaco na pilha */
    if (topo == tamanho) exit(0);
    /* Acrescenta o valor na pilha */
    pilha[topo++] = num;
  }
int PILHA::ret()
  {
    /* Verifica se a pilha nao esta vazia */
    if (topo == 0) exit(0);
    /*Retorna o elemento do topo */
    return(pilha[topo--]);
  }
```

Observe que os atributos de dados são acessados nas rotinas como se fossem variáveis globais. Isso só é possível porque as rotinas pertencem a classe onde estes atributos estão declarados (escopo de classe).

Para acessarmos os elementos de um objeto usamos a mesma sintaxe usada para acessar os membros de uma "**struct**". O programa mostrado anteriormente fica, então:

```
void main()
  {
    PILHA p1;
    int r;
    p1.init(50);
    for(r=0; r<50; r++)
      p1.ins(r);
    for(r=0; r<50; r++)
      printf("%d\n",p1.ret());
  }
```

Observe que o encapsulamento eliminou duas das regras de utilização das rotinas que implementavam a pilha sem o uso de classes:

- não existe outra alternativa para se acessar a estrutura de dados a não ser através das rotinas, pois os atributos de dados estão "escondidos" na parte privada da classe.
- não existe mais a necessidade de se passar uma referência à estrutura de dados que descreve a pilha como parâmetro para as rotinas, visto que agora essas estruturas são parte integrante do objeto.

Como vantagem adicional pode-se ainda simplificar o nome das rotinas uma vez que sabe-se que elas se referenciam a uma pilha pela classe a que pertence o objeto "p1".

Exercícios:

1) O fonte "exer4.cpp" contém o exemplo apresentado. Altere o programa:

   - declare um novo objeto pilha ;

   - desempilhe os valores armazenados no objeto "p1" e armazene-os no novo objeto;

   - imprima o conteúdo de ambas as pilhas.

2) Acrescente uma função membro à classe pilha que seja capaz de imprimir o conteúdo da pilha. Altere o programa para que passe a usar essa nova função.

5.2 Construtores e destrutores

O último dos problemas apresentados no início do ítem 5.1 é o fato de que o usuário de um objeto do tipo PILHA não pode esquecer nunca de ativar a rotina "init" sempre antes de usar efetivamente o objeto. Este fato pode levar a muitos erros de utilização.

Como é comum que um objeto requeira inicialização antes de ser usado, existem as funções construtoras. Uma função contrutora é uma função que possui o mesmo nome da classe e que é executada automaticamente quando da criação de um objeto. Um exemplo de função construtora para a classe PILHA poderia ser:

```
PILHA::PILHA()
  {    int r;
    /* Verifica o tamanho */
    tamanho = TAM_MAX_PILHA;
    /* Zera o vetor que serve de base a pilha */
    for(r=0; r<tamanho; r++)
      pilha[r] = 0;
    /* Inicializa o topo */
    topo = 0;
  }
```

Observe que neste exemplo foi retirado o parâmetro que indicava o tamanho da pilha sendo assumido o tamanho máximo. Mais adiante será abordado o tema dos parâmetros para as funções construtoras. Nota-se também que a função construtora não tem "tipo de retorno". Isso ocorre porque as funções construtoras não podem retornar valores.

Da mesma forma que a função construtora executa automaticamente quando um objeto é criado, a função destrutora executa automaticamente quando um objeto é destruído. A função destrutora tem o mesmo nome da classe antecedido do símbolo "~". Um exeplo de função destrutora para a classe PILHA pode ser:

```
PILHA::~PILHA()
  {
    printf("Pilha destruida\n");
  }
```

Exercícios:

1) Altere o fonte "exer4.cpp" incluindo as funções construtora e destrutora apresentadas.
2) Altere a classe PILHA de maneira que o vetor "pilha" seja alocado dinamicamente usando a função "malloc". Não esqueça de providenciar a desalocação do vetor na destrutora.
3) Crie várias pilhas no programa principal e observe o funcionamento das construtoras e destrutoras. Crie uma pilha dentro de um bloco de comandos condicional para verificar o escopo de existência do objeto.
4) Acrescente uma rotina à classe PILHA que seja capaz de imprimir as características da pilha: tamanho máximo e número de elementos. Use-a no programa.

5.3 Herança

A herança é uma das principais características da programação orientada a objetos. Através dela, é possível que uma classe transmita suas características a outras classes que são ditas derivadas ou especializadas de uma classe base, chamada "superclasse". Com isto temos a formação de uma estrutura hirárquica (hierarquia de classes) onde a superclasse possui as características comuns das suas subclasses. As características (dados e funções) da superclasse não precisam ser redefinidos nas subclasses pois estes serão "herdados". Para ver como funciona isto, vamos a um exemplo:

Temos uma superclasse (ou classe base) chamada "veículos de estrada" que possui características genéricas de todos os tipos de veículos, como seu número de rodas, quantidade de passageiros e operações sobre estes dados:

```
class veiculos_de_estrada
      {
        int rodas;
        int passageiros;
      public:
        void fixa_rodas(int num);
```

```
        int obtem_rodas(void);
        void fixa_passageiros(int num);
        int obtem_passageiros(void);
       };
```

As características acima são bastante genéricas e não servem para conter os dados de todos os tipos veículos. Por isto, são necessárias classes mais especializadas, aptas a particularizar os dados em função dos tipos de veículos que transitam na estrada. Por exemplo, se é necessário manter dados sobre caminhões, seria importante conter os dados genéricos de "veiculos_de_estrada", além de informações mais específicas como a sua carga transportável, por exemplo. Assim, abaixo é definida uma classe que herda todas as características de "veiculos_de_estrada" e acrescenta a carga:

```
class caminhao:public veiculos_de_estrada
       {
         int carga;
       public:
         void fixa_carga(int tamanho);
         int obtem_carga(void);
         void exibe(void);
        };
```

Da mesma forma, podemos particularizar as coisas para automóveis, criando uma classe que especifica de que tipo de automóvel estamos tratando:

```
enum type {carro, furgao, perua};
class automovel:public veiculos_de_estrada
       {
          enum type tipo_de_carro;
        public:
          void fixa_tipo(enum type t);
          enum type obtem_tipo(void);
          void exibe(void);
        };
```

Abaixo está um exemplo completo mostrando operações com herança de classes:

```
#include <iostream.h>
// Cria a Superclasse "veiculos_de_estrada":
class veiculos_de_estrada
      {
        int rodas;
        int passageiros;
      public:
        void fixa_rodas(int num);
        int obtem_rodas(void);
        void fixa_passageiros(int num);
        int obtem_passageiros(void);
       };
// Classe caminhao : Herda de "veiculos_de_estrada":
class caminhao:public veiculos_de_estrada
       {
         int carga;
       public:
         void fixa_carga(int tamanho);
         int obtem_carga(void);
         void exibe(void);
        };
enum type {carro, furgao, perua};
//Define a classe automovel, que tambem herda de "veiculos_de_estrada":
class automovel:public veiculos_de_estrada
       {
          enum type tipo_de_carro;
        public:
          void fixa_tipo(enum type t);
          enum type obtem_tipo(void);
```

```
           void exibe(void);
         };
void veiculos_de_estrada::fixa_rodas(int num)
{
 rodas = num;
}
int veiculos_de_estrada::obtem_rodas(void)
{
 return(rodas);
}
void veiculos_de_estrada::fixa_passageiros(int num)
{
 passageiros = num;
}
int veiculos_de_estrada::obtem_passageiros(void)
{
 return(passageiros);
}
void caminhao::fixa_carga(int tamanho)
{
 carga = tamanho;
}
void caminhao::exibe(void)
{
  cout << "rodas: " << obtem_rodas() << "\n";
  cout << "passageiros: " << obtem_passageiros() << "\n";
  cout << "capacidade de carga em m3: " << carga << "\n";
}
void automovel::fixa_tipo(enum type t)
{
 tipo_de_carro = t;
}
enum type automovel::obtem_tipo(void)
{
 return tipo_de_carro;
}
void automovel::exibe(void)
{
 cout << "rodas: " << obtem_rodas() << "\n";
 cout << "passageiros: " << obtem_passageiros() << "\n";
 cout << "tipo ";
 switch(obtem_tipo())
  {
   case furgao : cout << "furgao\n";
                 break;
   case carro  : cout << "carro\n";
                 break;
   case perua: cout << "perua\n";
                  break;
  }
}
void main()
{
 caminhao t1,t2;
 automovel c;
 t1.fixa_rodas(18);
 t1.fixa_passageiros(2);
 t1.fixa_carga(3200);
 t2.fixa_rodas(6);
 t2.fixa_passageiros(3);
```

```
 t2.fixa_carga(1200);
 t1.exibe();
 t2.exibe();
 c.fixa_rodas(4);
 c.fixa_passageiros(6);
 c.fixa_tipo(furgao);
 c.exibe();
}
```

A forma geral da herança é:

```
class nome_da_classe_derivada : acesso nome_da_classe_base
   {
    // Corpo da classe
   };
```

A definição de "acesso" é opcional. Se este está presente, entretanto, deverá ser ***public*** ou ***private***.

public ### Todos os elementos públicos da classe base se mantém públicos na classe derivada. No exemplo, "caminhao" e "automovel" possuem acesso às funções membros de "veiculo_de_estrada", mas não podem acessar os campos rodas e passageiros.

private ### Os membros públicos da classe base passam a ser privados na classe derivada. Assim, se esta classe derivada passar a ser base para outra classe a nova herdeira não mais terá acesso aos membros públicos da primeira classe base. ***Private*** é o valor "default" quando se omite o acesso na herança.

Exercícios:

1) O fonte "exer5.cpp" contém o exemplo acima. Crie a classe "motocicleta", derivada de "veiculos_de_estrada".
2) Crie uma superclasse chamada "veiculos", da qual "veiculos_de_estrada" passa a ser uma derivada. Crie também uma classe chamada "aviões", derivada de "veículos".
3) Crie funções construtoras para cada uma das classes de maneira que seja possível inicializar os objetos na declaração. Crie versões sobrecarregadas para essas construtoras (com e sem parâmetros) de forma que a inicialização seja opcional.
4) Crie instâncias das novas classes derivadas.

   Observe ainda nesses exercícios uma forma de "polimorfismo" semelhante a sobrecarga de funções vista no ítem 4 no caso da função "exibe". Cada objeto tem sua própria implementação para "exibe", embora semanticamente ambas façam a mesma coisa.
5) O fonte "exer6.cpp" contém a implementação da classe "RELOGIO" e da classe "RELOGIO_DIGITAL_TEXTO", derivada de relógio. A classe "RELOGIO" contém as rotinas que permitem acessar o relógio interno do PC através de rotinas próprias do "Borland C++". A classe "RELOGIO_DIGITAL_TEXTO" contém um conjunto de rotinas para a exibição da hora corrente em tela modo texto. A partir do entendimento do programa que usa esta hierarquia de classes, faça as seguintes modificações:
   a) Crie uma classe "DESPERTADOR", derivada de "RELOGIO_DIGITAL_TEXTO" que possua uma função "SetAlarme" que permita definir a hora e o minuto em que deve ser soado um alarme. Na construtora da classe deve ser assumido que o despertador deverá tocar as "00:00". Verifique que função da classe RELÓGIO_DIGITAL_TEXTO deve ser sobrecarregada na classe derivada para permitir o teste da hora do alarme. Utilize uma seqüência de "bips" do alto falante (printf("\a");) para simular a campainha do despertador.
   b) Sobrecarregue a função "SetAlarme" para que possua uma versão que selecione o alarme para tocar 1 minuto depois de sua ativação.
   c) Faça um programa que exiba instancias da classe "RELOGIO_DIGITAL_TEXTO" e "DESPERTADOR". Teste o uso das duas formas de ativar o alarme.
   d) Responda: em que momento um objeto da classe "RELOGIO_DIGITAL_TEXTO" tem o valor da hora atualizado de maneira a que o horário seja exibido coerentemente desde a primeira vez?

6. Construtores Parametrizados

No item 5.4 o uso de uma função construtora foi vantajoso na medida em que tornou automática a inicialização das estruturas que controlam a "PILHA". A desvantagem, entretanto, foi o fato de que o tamanho máximo da PILHA passou a ser fixo, visto que a função construtora não tinha parâmetros. As funções construtoras, embora não admitam retorno pelo nome da função, admitem lista de parâmetros como qualquer outra função. No caso a função construtora da classe PILHA ficaria:

```
PILHA::PILHA(int tam)
  {
    int r;
    if (tam >= TAM_MAX_PILHA) exit(0);
    else tamanho = tam;
    /* Verifica o tamanho */
    /* Zera o vetor que serve de base a pilha */
    for(r=0; r<tamanho; r++)
      pilha[r] = 0;
    /* Inicializa o topo */
    topo = 0;
  }
```

Para passar um parâmetro a uma função construtora deve-se associar os valores que estão sendo passados com o objeto quando ele esta sendo declarado. No caso o parâmetro ou parâmetros devem seguir o nome do objeto e estar entre parênteses.

Ex:

```
PILHA x(40);
PILHA p1(10), p2(67);
int x;

scanf("%d",&x);
PILHA pilha(x);
```

Quando se trabalha com herança, deve-se cuidar a passagem de parâmetros para **as funções construtoras**. A função construtora de uma classe derivada deve sempre providenciar a passagem dos parâmetros para a construtora da classe pai.

Ex:

```
class C1
  {
    int x,y;
    public:
    C1(int px,int py)
      {
       x = px;
       y = py;
      }
  };
class C2 : public C1
  {
    int b,c,d;
    public:
    C2(int a);
  };
C2::C2(int a)
      :C1(a,a*2)
  {
    b = a;
    c = a*2;
    d = 23;
  }
```

```
void main()
  {
    C2 n(12);
  }
```

Exercícios:

1) Analise o fonte "exer7.cpp". Compile-o e verifique (depurador - opção debug|inspec) a passagem dos parâmetros para as funções construtoras e a hierarquia de herança utilizada.
2) Altere a construtora da classe relógio de maneira que a mesma possua um parâmetro que especifica se a mesma deve emitir um "bip" ou não quando um objeto "RELOGIO" for criado (não use argumentos default).
3) Crie a classe "RELOGIO_ANALOGICO_COLORIDO" derivada de "RELOGIO_ANALOGICO". A construtora desta classe deve ter um parâmetro a mais que indica a cor dos ponteiros. Observe que esta classe necessita apenas de um atributo de dados para armazenar a cor dos ponteiros, da rotina construtora e de uma nova versão para a rotina "atualiza_mostrador". Crie uma instância desta nova classe no "main" e integre o funcionamento deste novo relógio com os demais.
4) Altere uma das classes do programa "exer7.cpp" de maneira que toda a vez que um relógio for destruído ele exiba a mensagem "Relogio destruido".
5) O que deve ser acrescentado para que a mensagem exibida na destrutora criada no exercício anterior possa indicar que relógio foi destruido e a que classe ele pertence?

7. Alocação dinâmica de memória

7.1 Alocação dinâmica de vetores

Um vetor é uma coleção de dados do mesmo tipo agrupados sob um único nome.

Em "C++" declara-se um vetor alocando-se uma região de memória suficiente para armazenar a quantidade de elementos de um dado tipo que se pretende. O endereço inicial desta área é armazenado em uma variável ponteiro.

Para armazenar uma lista de 10 valores inteiros é necessário:

a) declarar uma variável que seja ponteiro para inteiro;

b) alocar espaço de memória suficiente para armazenar 10 inteiros;

c) armazenar o endereço inicial da área alocada em uma variável do tipo ponteiro.

Exemplo:

```
#include <stdio.h>
void main()
  {
     int *vet,i;
     vet = new int[10];
     if (vet == NULL)
       {
          printf("Falta memória para a alocação do vetor\n");
          exit(1);
       }
     for(i=0; i<10; i++)
       vet[i] = i*2;
  }
```

O operador "**new**" aloca uma região de memória para conter a estrutura de dados solicitada. Pode ser usada tanto para alocação de vetores como de variáveis simples ou objetos. No caso de alocação de vetores deve ser especificado o número de posições do vetor entre colchetes após o tipo de dados base.

A liberação da área de memória alocada com "**new**" deve ser feita com o uso do operador "**delete**". Áreas alocadas com "**new**" não são liberadas após o término da função onde foram alocadas (mas o ponteiro sim !!!). As áreas só são automaticamente liberadas após o término do programa.

à semelhança da linguagem C, caso não exista espaço disponível na memória para a alocação solicitada, "**new**" retorna zero ou NULL (constante definida em <stdlib.h>). É extremamente importante testar se a alocação foi bem sucedida, caso contrário corre-se o risco de se tentar trabalhar sobre uma estrutura que não existe.

É possível também definir uma função para tratamento dos casos de falta de memória. Isso é feito informando o nome da função de tratamento para a função "set_new_handler" definida em "new.h".

Exemplo:

```
#include <new.h>
#include <stdlib.h>
#include <iostream.h>
void erro_mem()
  {
     printf("Falta de memória\n");
     exit(0);
  }
void main()
  {
     int *vet,i;
     set_new_handler(erro_mem);
     vet = new int[10];
```

```
    for(i=0; i<10; i++)
      vet[i] = i*2;
    delete vet;
  }
```

- **Atenção:** o operador "**delete**" não zera o ponteiro, apenas libera a área. É uma boa prática de programação atribuir NULL aos ponteiros após a liberação.

Exercício:

1) Altere o fonte "exer4.cpp" de maneira que o objeto pilha receba por parâmetro, na construtora, o tamanho da pilha. Providencie para que a alocação do vetor na construtora seja feita com o comando "**new**" e a liberação, na destrutora, pelo comando "**delete**". Crie uma função de tratamento de erros e acrescente no "main" uma referência para esta função através da função "set_new_handler".
2) Execute o programa do exercício 1 utilizando um tamanho absurdo para a pilha, de maneira a forçar um erro de falta de memória. Atenção: cuidado com o tipo da variável usada para indicar o tamanho da pilha: "**int**" x "long".

7.2 Alocação dinâmica de objetos

A única diferença entre a alocação dinâmica de uma variável qualquer e a alocação dinâmica de um objeto, é que no caso de objetos a utilização do operador "**new**" implicará na ativação da função construtora e o uso do operador "**delete**" na ativação da função destrutora.

No caso de alocação dinâmica de vetores deve-se ter o cuidado de que a função construtora do objeto não tenha parâmetros ou que pelo menos exista uma versão sem parâmetros.

A deleção de um vetor, usando o comando "**delete**", necessita que se indique quantas vezes a função destrutora da classe deverá ser ativada.

Ex:

```
CLASS1 * vet;
.
.
vet = new CLASS1[50];
.
.
delete [50]vet;
```

Os compiladores mais recentes da Borland emitem um "warning" avisando que não é necessário indicar a quantidade de posições do vetor na deleção. Quando não se indica, porém, o funcionamento é incorreto.

Exemplo:

```
#include <stdlib.h>
#include <iostream.h>
#include <string.h>
#include <stdio.h>
class teste
  {
     char nome[80];
     public:
     teste(void);
     ~teste(void);
     void setnome(char *);
     char *readnome(void);
  };
teste::teste()
  {
     strcpy(nome,"vazio");
  }
teste::~teste()
  {
```

```
      printf("Liberando nome: %s\n",nome);
  }
void teste::setnome(char *n)
  {
     if (strlen(n) < 80) strcpy(nome,n);
  }
char *teste::readnome()
  {
     return(nome);
  }
void main()
  {
     teste *vet;
     int r,n;
     char aux[80];

     printf("Quantos nomes? ->");
     scanf("%d",&n);
     vet = new teste[n];
     if (vet == NULL) exit(0);
     printf("\n");
     for(r=0; r<n; r++)
        printf("%s\n",vet[r].readnome());
     for(r=0; r<n; r++)
       {
              printf("Digite o nome %d: ");
              gets(aux);
              vet[r].setnome(aux);
              printf("\n");
       }
     for(r=0; r<n; r++)
        printf("%s\n",vet[r].readnome());
     delete [n]vet;
  }
```

Exercícios:

1) Analise o fonte "exerc8.cpp". Construa um programa que pergunte o número de funcionários de uma empresa, aloque um vetor com uma posição para cada funcionário e solicite e armazene os dados dos funcionários. O programa deve, em seguida, imprimir a lista dos funcionários que ganham muito (acima de R$ 9000,00).

   Obs:

   Aloque o vetor usando o comando "**new**"

   Destrua o vetor usando o comando "**delete**"

2) É possível fazer o exercício anterior sem usar alocação dinâmica de memória ("**new**" e "**delete**") ? Como? Qual a diferença básica?

7.3 Ponteiros para objetos

O acesso normal aos atributos e métodos de um objeto é feito com auxílio do operador ".". O que muda quando trabalhamos com ponteiros é a substituição do "." pelo operador "->".

Exemplo:

Obs: este exemplo se baseia na classe "teste" definida no item 7.2.

```
void main()
  {
     teste *ptobj;
     char aux[80];
     ptobj = new teste;
```

```
    if (ptobj == NULL) exit(0);
    printf("\n");
    printf("Digite um nome: ");
    gets(aux);
    ptobj->setnome(aux);
    printf("\n%s\n",ptobj->readnome());
    delete vet;
}
```

Exercícios:

1) Altere o fonte "exer7.cpp" de maneira que todos os relógios sejam alocados dinamicamente. Faça as modificações necessárias no "main" para tornar o acesso as propriedades desses objetos compatíveis com as alterações.

8. Class X struct X union

C++ extende o sistema de estruturas da linguagem C padrão, permitindo que uma estrutura possua funções, como a seguir:

```
struct exemplo
   {
      void coloca(int a, int b)
          {
              x=a; y=b;
          }
      void mostra()
          {
              printf"%d e %d\n", x y);
          }
     private:
     int x, y;
  };
```

Assim, uma **struct** é exatamente igual a uma classe. A única diferença é que o direito de proteção "default" é **public**, ao contrário da classe onde é **private**. Por isto, na definição acima, x e y foram colocados como **private** para manter o encapsulamento dos objetos do tipo "exemplo". A definição é equivalente a:

```
class exemplo
   {
       int x, y;
       public:
       void coloca(int a, int b)
         {
             x=a; y=b;
         }
       void mostra()
         {
              printf("%d e %d\n",x,y);
         }
   };
```

No caso de "unions" vale a mesma regra. Uma "union" é uma classe onde todos os elementos são públicos por "default" e onde todos os atributos de dados compartilham o mesmo espaço de memória.

Exercício:

1) Analise, compile e execute o fonte "exer9.cpp".

9. Herança múltipla:

É possível fazer com que uma classe herde atributos de mais de uma classe. Para isto, se usa uma lista de entradas, separadas por vírgulas, de classes com seus respectivos direitos de acesso, na lista de classes base da classe derivada. O formato é:

```
class clase_derivada: acesso classe_base1, acesso classe_base2,...
```

No exemplo a seguir, os atributos das classes bases são de tipo **protected**. Isto significa que para as demais classes os atributos são privados, porém eles são públicos para as classes derivadas.

```
//
//Exemplo de heranca multipla
//
#include <iostream.h>
class X
    {
       protected:
       int a;
       public:
       void make_a(int i);
     };
class Y
     {
        protected:
        int b;
        public:
        void make_b(int i);
      };
// Z herda de X e Y
class Z : public X, public Y
  {
     public:
     int make_ab(void);
   };
void X::make_a(int i)
  {
     a = i;
  }
void Y::make_b(int i)
  {
     b = i;
  }
int Z::make_ab(void)
  {//pode acessar a e b porque eh "protected"
     return a*b;
  }
void main(void)
  {
     Z i;
     i.make_a(10);
     i.make_b(12);
     printf("%d\n", i.make_ab());
  }
```

Quando a classe base possui uma construtor com parâmetros, é necessário que suas derivadas também possuam construtores para que os parâmetros da construção da classe base sejam passados. Como no exemplo:

```
//
// Herancas onde a classe base possui construtor com
// parametros
```

```
#include <iostream.h>
class X
    {
        protected:
        int a;
        public:
        X(int i); //construtor com parametro
    };
class Y
    {
        protected:
        int b;

        public:
        Y(int i);
    };
//Z herda de X e Y
class Z: public X, public Y
    {
         public:
         Z(int x, int y);
         int make_ab(void);
    };
X::X(int i)
    {//construtor de X
         a = i;
         printf("Inicializando X\n");
    }
Y::Y(int i)
    {//construtor de Y
         b = i;
         printf("Inicializando Y\n");
    }
//Inicializa X e Y por meio do construtor de Z.
//Note que Z nao usa x e y nesse exemplo, mas
//poderia, se fosse necessario
Z::Z(int x, int y) : public X(x), public Y(y)
    {
        printf("Inicializando Z\n");
    }
int Z::make_ab(void)
   {
         return a*b;
   }
void main(void)
    {
      //i e' objeto da classe Z e 10,20 sao os parametros da construcao
        Z i(10,20);
        printf("%d", i.make_ab());
    }
```

Exercícios:

1) O fonte "exer12.cpp" possui a definição de uma classe "SOCIO" e de uma classe "ATLETA". Defina uma classe "SOCIO_ATLETA" derivada de ambas. Instancie objetos das três classes e imprima seu conteúdo.
2) Qual a diferença, neste caso, entre definir "SOCIO_ATLETA" como uma hierarquia ou como uma agragação?

10. Agregações

A grande vantagem de se definir agregações é que se consegue definir relacionamentos do tipo "é parte de". Um exemplo pode ser um microcomputador composto de gabinete, teclado e vídeo. Cada uma dessas partes é um objeto individual. Agregadas, formam um objeto mais complexo.

A definição de agregações em "C++" é feita pela definição de atributos que são objetos. Esses podem ser definidos diretamente na classe ou alocados dinamicamente a partir de atributos que são ponteiros. O que se torna relevante nesses casos é saber como se comportam as funções construtoras e como passar os parâmetros para as mesmas.

Exemplo:

```
#include <stdlib.h>
#include <iostream.h>
#include <string.h>
#include <stdio.h>
class video
  {
    char marca[80];
    public:
    video(char *m) { strcpy(marca,m);}
    ~video() { printf("Destruiu o video\n"); }
    void imp(void);
  };
class teclado
  {
    char descricao[100];
    public:
    teclado(char *descr) { strcpy(descricao,descr);}
    ~teclado() { printf(""Destruiu teclado\n"; }
    void imp(void);
  };
class gabinete
  {
    char cpu[80];
    public:
    gabinete(char *c) { strcpy(cpu,c); }
    ~gabinete() { printf(""Destruiu gabinete\n"; }
    void imp(void);
  };
class micro
  {
    char nome[80];
    video *vid;
    teclado t;
    gabinete *gab;
    public:
    micro(char *nome,char *tipo_cpu,char *marca_video);
    ~micro(void);
    void imp(void);
  };
// Implementacao da classe video
void video::imp()
  {
    printf(""Marca do video: " << marca << "\n");
  }
// Implementacao da classe teclado
void teclado::imp()
  {
```

```
    printf("Descricao do teclado: " << descricao << "\n");
  }
// Implementacao da classe gabinete
void gabinete::imp()
  {
    printf("Tipo da cpu: " << cpu << "\n");
  }
// Implementacao da classe micro
micro::micro(char *n,char *tipo_cpu,char *marca_video)
          :t("Teclado comum - 128 teclas")
  {
    strcpy(nome,n);
    vid = new video(marca_video);
    gab = new gabinete(tipo_cpu);
  }
micro::~micro()
  {
    delete vid;
    delete gab;
    printf(""Destruiu micro:" << nome << "\n";
  }
void micro::imp()
  {
    printf(""\nDescricao do micro:" << nome << "\n";
    vid->imp();
    t.imp();
    gab->imp();
    printf(""\n";
  }
// Rotina principal
void main()
  {
    micro *micro1;
    micro micro2("Micro2","386 DX 25","Nec MultiSync 3D");
    micro1 = new micro("Micro1","486 DX 66","Samsung SyncMaster3");
    micro1->imp();
    micro2.imp();
    delete micro1;
  }
```

Exercício: *(os exercícios referêm-se ao fonte "exer11.cpp")*

1) Altere a construtora da classe "CASA" de maneira que ela possa receber como parâmetro a situação das portas e janelas.
2) Crie uma função para a classe casa que seja capaz de imprimir na tela a situação das portas e janelas de uma casa.
3) Acrescente no "main" um trecho onde seja permitido ao usuário alterar a situação das portas e janelas de uma das casas.
4) Crie diversas instâncias de casas e imprima a situação das suas portas e janelas.
5) Responda:

   a) Qual seria a diferença se a agregação "CASA" fosse montada com os componentes definidos diretamente como objetos e não alocados dinamicamente através de atributos ponteiros. Procure esboçar através de um desenho a estrutura de memória de um objeto "CASA" nos dois casos.

   b) Como ficaria a passagem de parâmetros para as funções construtoras dos componentes caso eles não fossem alocados dinamicamente? Que método você considera mais vantajoso neste caso.

   c) Qual seria a vantagem se a classe "CASA" também fosse derivada de "BOX"?

6) Crie uma classe "RUA" que seja uma agregação de "CASAs". A classe deve possui um vetor do tipo "CASA" cujo tamanho é definido na construtora. Observe que a criação de um vetor de objetos implica em que o mesmo não possua construtora com parâmetros. O que deve ser alterado na classe casa, então?

7) Acrescente um atributo "visivel" na classe "CASA" e em cada uma das componentes "PORTA", "JANELA" e "BOX". O objetivo deste atributo é indicar quando um objeto esta sendo exibido na tela. Desta forma as funções do tipo "SetCor" ou "SetEstado" teriam condições de re-exibir os elementos necessários.

8) Acrescente comandos de impressão nas rotinas destrutoras de forma a ser possível observar a ordem em que as mesmas são ativadas.

11. Membros estáticos em classes:

Quando for necessário que todos os objetos de uma mesma classe compartilhem uma única variável, podemos definir uma variável membro com o atributo **static**, que fará com que todos os objetos compartilhem uma única cópia da variável. Veja o exemplo:

```
class contador
  {
    static int cont;
    public:
    void seta_contador(int i)
        {
          cont = i;
        }
    void mostra_contador()
        {
         printf("Valor do contador = %d \n",cont);
        }
  };
```

Neste caso, a variável cont existirá apenas uma vez, não interessando a quantidade de objetos do tipo contador que sejam criados. A declaração "**static int** cont", porém, não cria área de armazenamento para a variável, sendo necessário abrir o seu espaço fora da classe:

```
//aloca area da variavel estatica:
int contador::cont;
```

Abaixo um exemplo completo:

```
#include <iostream.h>
class contador
  { //para todos os objetos so existe
    //um unico "cont"
    static int cont;
    public:
    void seta_contador(int i)
      {
        cont = i;
      }
    void mostra_contador()
      {
        cout << "valor do contador = ";
        cout << cont << "\n";
      }
  };
//aloca area da variavel estatica:
int contador::cont;
void main()
{
 contador a,b;
 a.seta_contador(4);
 a.mostra_contador();
 b.mostra_contador();
 b.seta_contador(5);
 b.mostra_contador();
 a.mostra_contador();
}
```

Funções membros também podem ser **static**. Neste caso o ponteiro "**this**" não é passado à função, e esta terá acesso apenas a variáveis **static** da classe e a outras funções membros também **static**.

Exercício:

1) Acrescente à classe "VETOR" uma rotina que seja capaz de infomar a qualquer momento quantas operações com vetores já foram feitas até o momento.

12. Passagem de parâmetros por referência

A linguagem "C" não permite passagem de valores por referência. só é admitida passagem de tipos escalares de dados por valor. Toda a passagem por referência é simulada através do uso de ponteiros.

A linguagem "C++" admite passagem de parâmetros por referência tanto de tipos escalares como de objetos. Ponteiros não podem ser passados por referência. Para declararmos um parâmetro por referência basta utilizarmos o operador "&" antes do nome do parâmetro. Outra característica importante da linguagem "C++" é que além de passagem por referência de objetos é possível também fazer passagem por valor (em "C" a passagem por valor é admitida somente para tipos escalares). A passagem de parâmetros através de ponteiros no estilo da linguagem "C" também é válida em "C++".

Exemplo:

```
void soma_sub(int a, int b, int &soma, int &sub)
  {
     soma = a + b;
     sub = a - b;
  }
void main()
  {
     int v1,v2,s,sub;
     v1 = 4;
     v2 = 9;
     soma_sub(v1,v2,s,sub);
     printf("%d %d",s,sub);
  }
```

É possível, também declarar variáveis "referências". Variáveis referências devem ser inicializadas na criação.

Ex:

```
void f()
  {
    int i;
    int &r = i;      // r referencia i
    r = 1;           // o valor de i torna-se 1
    int *p = &r;     // p aponta para i
    int &rr = r;     // rr referencia o que r referencia, ou seja i
  }
```

Uma referência não pode ser alterada para referenciar outro objeto após a inicialização. Por esta razão referências não são muito úteis em "C++" a não ser para simplificar a passagem de parâmetros. Não existiria razão para complicar a linguagem aumentando a flexibilidade das referências uma vez que todas as vantagens eventualmente obtidas podem ser conseguidas com ponteiros.

Exercícios:

1) Digite o exemplo da página anterior e verifique no depurador seu funcionamento.
2) Acrescente uma variável inteira "b" e tente armazenar uma sua referência na variável "r". Explique qual a razão da mensagem de erro.

13. Parâmetros e ponteiros

Um objeto pode ser passado por parâmetro de diversas formas em "C++". Em cada caso é preciso observar quando são criadas cópias dos objetos ou quando apenas referências são passadas. É importante prestar atenção também quando as funções construtoras e destrutoras são ativadas e desativadas.

Exemplo:

Obs: este exemplo se baseia na classe teste definida no item 7.2.

```
teste *alocaobjpt()
  {
    teste *pt;
    pt = new teste;
    pt->setnome("Com ponteiro");
    return(pt);
  }
void naoinstancia(teste obj)
  {
    obj.setnome("Recebe por valor e nao retorna");
  }
teste instancia(teste obj)
  {
    obj.setnome("recebe e retorna por valor");
    return(obj);
  }
teste instanciaobj()
  {
    teste obj;
    obj.setnome("Retornando copia");
    return(obj);
  }
void instanciaobjref(teste &obj)
  {
    obj.setnome("Por referencia explicita");
  }
void instanciaobjrefpt(teste *obj)
  {
    obj->setnome("Por referencia usando ponteiros");
  }
void main()
  {
    teste *ptobj,obj1,obj2,obj3,obj4,obj5;
    clrscr();
    ptobj = alocaobjpt();
    naoinstancia(obj1);
    obj2 = instancia(obj2);
    obj3  = instanciaobj();
    instanciaobjref(obj4);
    instanciaobjrefpt(&obj5);
    printf("\n");
    printf("%s\n",ptobj->readnome());
    printf("%s\n",obj1.readnome());
    printf("%s\n",obj2.readnome());
    printf("%s\n",obj3.readnome());
    printf("%s\n",obj4.readnome());
    printf("%s\n",obj5.readnome());
    printf("\n");
    delete ptobj;
  }
```

Para acompanhar a análise feita a seguir compile o fonte "exer9.cpp" e execute-o passo a passo com o depurador usando a opção "inspect" e a janela "watch".

Vamos analisar inicialmente as declarações feitas na função "main". As variáveis ptobj, obj1, obj2, obj3, obj4 e obj5 são todas locais a função "main" e por esta razão são alocadas na pilha quando a função se inicia e desalocadas quando a função se encerra. A variável ptobj é a única que é um ponteiro. As outras três são declarações de objetos, portanto a função construtora é ativada quando da alocação destes na pilha e a destrutora quando de sua desalocação no encerramento da função. O ponteiro ptobj apontará para um objeto alocado e desalocado com os operadores "**new**" e "**delete**". Analisemos agora as funções.

A função "alocaobjpt" aloca um objeto com auxílio de um ponteiro local. O construtor do objeto é acionado quando da execução do operador "**new**". Observe que ao término da função o ponteiro local é destruído, porém, o objeto alocado não. O objeto só será destruído (e sua destrutora ativada) quando da execução do operador "**delete**" na "main". Objetos ou vetores alocados com o operador "**new**" não possuem sua existência atrelada a funções, mas sim ao controle do usuário ou término do programa. A função retorna o endereço do objeto alocado pelo comando "**return**". **Atenção:** deve-se ter o cuidado para **não retornar endereço de variáveis locais**.

A função "naoinstancia" recebe uma cópia do objeto (passagem por valor). Por esta razão as alterações feitas no atributo nome não serão retornadas a função "main". Observe a ativação da função construtora quando a função se inicia (construção da cópia temporária) e da destrutora no término da função quando da liberação da cópia.

A função "instancia" retorna as alterações feitas na cópia do objeto recebido fazendo um retorno por valor. Observe que para tanto necessita criar uma nova cópia na pilha da mesma forma que a função "instanciaobj".

A função "instanciaobj", por outro lado, utiliza apenas retorno "por valor". Um objeto local é definido para o instanciamento do objeto e, em seguida, uma cópia do mesmo é retornada com o comando "**return**". Observe, durante a execução, que o destrutor será chamado duas vezes ao término da função: uma para destruir a variável local e outra para destruir a cópia temporária alocada na pilha para ser devolvida pelo comando "**return**". Dependendo do compilador essa dupla alocação pode ser otimizada.

A função "instanciaobjref" se vale da passagem de parâmetro por referência do "C++" para alterar o objeto obj2 declarado na função "main". Observe que desta forma não é gasto espaço na pilha com a alocação de cópias do objeto pois apenas uma referência é passada.

A função "instanciaobjrefpt" trabalha, em princípio, da mesma forma que a anterior. A diferença é que em vez de se utilizar da passagem por referência do "C++" se utiliza dos ponteiros para obter o mesmo resultado. Esse tipo de procedimento é bastante utilizado na linguagem "C" onde não existe passagem por referência.

A escolha da maneira pela qual os objetos devem ser alocados e passados como parâmetro irá depender muito da aplicação. A alocação de muitas varáveis locais irá implicar em tempo na chamada da função e ocupação da pilha. Tem a vantagem de prover desalocação automática. A passagem por referência (por qualquer um dos métodos) deve ser usada quando não comprometer a estruturação ou o encapsulamento dos objetos por evitar a alocação de cópias desnecessárias na pilha e, conseqüentemente, ser mais rápida.

Questões:

1) Porque não se pode usar retorno por referência (por qualquer método) na função "naoinstancia"?
2) Porque não se pode usar retorno por referência na função "instanciaobj"?
3) Porque a diferença no acesso aos elementos dos objetos nas funções "instanciaobjref" e "instanciaobjrefpt"?

14. Funções e classes "Friend" (amigas)

Uma função que não é membro de uma classe pode acessar seus componentes privados se for declarada como "**friend**" da classe. No exemplo abaixo, é declarada uma função chamada "amiga" que pode acessar os membros internos da classe "Klase":

```
class Klase
  {
    ....
    ....
   public:
    friend void amiga();
    ....
   };

void amiga()
{
  //aqui vai o corpo da função
 }
```

A função "amiga()" não pertence à "Klase". Ela, porém, é uma função comum com direito de acesso aos componentes internos da classe onde foi referida como "**friend**".

A grande utilidade de funções "**friend**" é permitir a comparação de valores pertencentes a duas classes distintas. Por exemplo, suponha a existência de duas classes: uma com informações sobre aviões e outra com informações sobre automóveis. Se as classes possuem informações de número de passageiros, por exemplo, poderia surgir a necessidade de comparar a capacidade de determinado automóvel com a de um avião específico. Para efetuar esta comparação seria necessária uma função com acessos aos dados privados de ambos os tipos (automóveis e aviões), como abaixo:

```
class aviao;  //Declaracao antecipada
class automovel
  {
      int potencia;
      int numero_de_passageiros;
      float preco;
    public:
      automoveis(int potencia, int num_pass, float preco);
      void aumenta_preco(float valor_acrescido);
      friend int capacidade_igual(automovel a, aviao b);
    };
class aviao
   {
      int empuxo;
      int numero_passageiros;
      char companhia[20];
    public:
      aviao(int empuxo, int capacidade);
      void poe_companhia(char comp[20]);
      friend int capacidade_igual(automovel a, aviao b);
    };

   //Abaixo o codigo da funcao "friend"
   int capacidade_igual(automovel a, aviao b)
   {
     if(a.numero_de_passageiros == b.numero_passageiros)
         return(1);
     else
         return(0);
   }
```

A declaração antecipada ("**class** aviao;") é necessária antes da declaração da classe automóvel pois esta contém a declaração da função "**friend**" que utiliza o tipo "aviao" como um de seus parâmetros. Assim, esta declaração está avisando ao compilador que a classe "aviao" existe e será completamente definida mais tarde.

Observação:

A função "**friend**" NÃO é membro da classe onde foi declarada. Assim, seria ilegal fazer como segue:

```
aviao boeing;
automóvel fusca, passat;
fusca.capacidade_igual(passat, boeing);
                  //Erro !! "capacidade_igual()"  não é membro!
```

O correto é:

```
capacidade_igual(passat, boeing);
               //Certo, esta forma é a válida
```

O exemplo a seguir mostra uma função "**friend**" que compara as cores de dois entes geométricos:

```
//
// Uso de funcoes "Friend"
//
#include <iostream.h>
#include <conio.h>
//Faz a declaracao da classe "linha" de forma antecipada porque e'
// necessaria no parametro da funcao "friend"
class linha;  //Declaracao antecipada
class box
         { //Dados e operacoes para retangulos
           int cor;
           int upx, upy;   //canto superior esquerdo
           int lowx, lowy; //canto inferior direiro
          public:
           friend int mesma_cor(linha l, box b);
           void indica_cor(int c);
           void define_box(int x1, int y1, int x2, int y2);
           void exibe_box(void);
          };
class linha
         { //Dados e operacoes para linhas
           int cor;
           int comecox, comecoy;
           int tamanho;
          public:
           friend int mesma_cor(linha l, box b);
           void indica_cor(int c);
           void define_linha(int x, int y, int l);
           void exibe_linha();
          };
// A funcao AMIGA de box e linha:
int mesma_cor(linha l, box b)
{
 if(l.cor == b.cor) return 1;
 return 0;
}
void box::indica_cor(int c)
{
 cor = c;
 }
void box::define_box(int x1, int y1, int x2, int y2)
{
 upx = x1;
```

```
 upy = y1;
 lowx = x2;
 lowy = y2;
}
void box::exibe_box(void)
{
 int i;
 textcolor(cor);
 gotoxy(upx, upy);
 for(i=upx; i<=lowx; i++)
  cprintf("-");
 gotoxy(upx, lowy-1);
 for(i=upx; i<=lowx; i++)
  cprintf("-");
 gotoxy(upx, upy);
 for(i=upy; i<=lowy; i++)
  {
   cprintf("|");
   gotoxy(upx, i);
   }
 gotoxy(lowx, upy);
 for(i=upy; i<=lowy; i++)
 {
  cprintf("|");
  gotoxy(lowx, i);
  }
}
void linha::indica_cor(int c)
{
 cor = c;
}
void linha::define_linha(int x, int y, int l)
{
 comecox = x;
 comecoy = y;
 tamanho = l;
}
void linha::exibe_linha(void)
{
 int i;
 textcolor(cor);
 for(i=0; i<tamanho; i++)
   cprintf("-");
}
main(void)
{
 box b;
 linha l;
 b.define_box(10, 10, 15, 15);
 b.indica_cor(3);
 b.exibe_box();
 l.define_linha(2, 2, 10);
 l.indica_cor(2);
 l.exibe_linha();
 if(!mesma_cor(l, b))
  printf("Nao sao da mesma cor\n");
 printf("pressione uma tecla\n");
 getch();
 //torna linha e box da mesma cor
 l.define_linha(2, 2, 10);
```

```
 l.indica_cor(3);
 l.exibe_linha();
 if(mesma_cor(l, b))
  printf(""Sao da mesma cor\n");
 getch();
 }
```

Além de "funções amigas", uma classe pode conter também "classes amigas". Uma classe amiga pode ter acesso aos dados privados da classe que a declarou como amiga. A declaração "**friend**", embora em muitas situações seja um recurso necessário, deve ser usada com cuidado visto que é uma forma de se abrir exceções no "escudo" do encapsulamento das classes.

Exemplo de declaração de classe amiga:

```
class ClasseY;
class ClasseX
  {
    int a,b;
    public:
    friend class ClasseY;
    ...
  };
class ClasseY
  {
    int z;
    public:
    void f1( ClasseX p)
      {
          z = p.a+p.b;
      }
  };
```

Exercícios:

1) Utilizando o fonte "exer11.cpp" crie uma função amiga das classes "JANELA" e "PORTA" que devolva o valor "1" quando ambas estiverem na mesma "situação" (ABERTA ou FECHADA) e o valor "0" caso contrário.
2) Para poder testar a função amiga definida no exercício 1, defina funções para a classe "CASA" que devolvam ponteiros para sua porta e janelas.
3) Defina uma classe amiga da classe "CASA" que possua uma rotina para aplicar uma translação sobre um objeto "CASA" recebido por parâmetro.

15. Auto-referência em objetos - Palavra reservada *this*:

Cada vez que se chama uma função membro de um objeto, automaticamente é passado um ponteiro para o objeto que a invocou. Este ponteiro pode ser acessado pela palavra reservada **this**. Este ponteiro é passado de forma automática a uma função membro quando esta é chamada. Ou seja, o ponteiro "**this**" é um parâmetro de todas as chamadas de funções membros.

Uma função pode acessar de forma direta os dados privados de sua classe. Por exemplo, dada a classe:

```
class cl
      {
         int i;
         ...
       };
```

Uma função membro pode assinalar um valor a i, fazendo: i = 10;

Em realidade, isto é uma abreviatura de **this**->i = 10;

Para ver como "**this**" opera, veja o exemplo:

```
class cl
   {
      int i;
      public:
      void load_i(int val)
        {
          this->i = val; // O mesmo que i=val
        }
      int get_i()
        {
          return(this->i); //O mesmo que return(i)
        }
   };
void main()
{
 cl obj;
 obj.load_i(10);
 printf("%d",obj.get_i());
}
```

Como visto, o uso de **this** neste contexto é dispensável. Porém ele é importante quando se trabalha com sobrecarga de operadores, como será visto no próximo item e em certos casos quando se deseja manter em um objeto uma referência para o objeto que o alocou.

Exercício:

1) Analise o fonte "exer18.cpp". Compile-o e execute-o.
2) Acrescente uma função na classe "CARRO" que retorne o imposto a pagar do veículo. A rotina recebe por parâmetro a taxa de imposto do carro ano 1995 e desconta 10% para cada ano de uso. Se por acaso o proprietário do carro tiver mais de 75 anos, o carro fica isento de imposto.
3) Acrescente na classe "PESSOA" uma rotina que devolve o endereço do "proprietario" correspondente. Acrescente no "main" uma chamada para esta função e utilize o endereço retornado para imprimir os dados do proprietário.
4) O que aconteceria se eliminassemos a função destrutora da classe "CARRO"? Que cuidados devem ser tomados se a destrutora for mantida e for acrescentada a rotina proposta no exercício 3.

16. Polimorfismo:

O polimorfismo é um conceito fundamental na programação orientada a objetos. Em C++, o polimorfismo é basicamente o processo pelo qual diferentes implementações de uma mesma função podem ser acessadas pelo mesmo nome. Por isto, o polimorfismo se caracteriza por "uma interface - múltiplos métodos". Isto significa que se pode acessar da mesma maneira uma classe geral de operadores, ainda que as ações específicas as quais estão associadas a cada operação possam ser diferentes.

Em C++, o polimorfismo é suportado tanto em tempo de compilação como em tempo de execução. A sobrecarga de operadores e funções é um exemplo de polimorfismo em tempo de compilação. Porém, somente sobrecarga de operadores e funções não pode efetuar todas as tarefas requeridas em uma verdadeira linguagem orientada a objetos. Por isto, C++ também provê polimorfismo em tempo de execução, mediante o uso de classes derivadas e de funções virtuais.

16.1 - Ponteiros para tipos derivados:

Quando existe uma classe base e uma classe derivada, é possível a um ponteiro do tipo da classe base apontar para elementos da classe derivada. Este, porém, na classe derivada, terá acesso somente aos atributos que foram herdados da classe base. No exemplo a seguir é mostrada uma classe base com nomes de pessoas e uma classe derivada que acrescenta o atributo número de telefone, dando origem a objetos com nomes de pessoas e seu respectivo número de telefone.

```
//Exemplo de ponteiros para tipos derivados
#include <iostream.h>
#include <string.h>
//construcao da classe principal:
class base
      {
        char nome[80];
       public:
        void coloca_nome(char *s)
             {
              strcpy(nome,s);
              }
        void mostra_nome()
             {
              cout << nome << "\n";
              }
       };
//contrucao da classe derivada
class derivada:public base
      {
        char telefone[80];
       public:
        void coloca_telefone(char *n)
             {
              strcpy(telefone, n);
              }
        void mostra_telefone()
             {
              cout << telefone << "\n";
              }
       };
void main()
{
 base *p;  //Este ponteiro acessa tanto "base"
           //quando "derivada"(so na parte comum)
 base obj_b;
 derivada *pd; //So acessa a classe derivada
 derivada obj_d;
 p = &obj_b; //aponta para obj. de base
```

```
 p->coloca_nome("Jose da Silva"); //acesso por ponteiro
 //acessa a classe derivada via ponteiro:
 p = &obj_d;
 p->coloca_nome("Andre da Silva");
 //mostra que nomes estao nos lugares certos:
 obj_b.mostra_nome();
 obj_d.mostra_nome();
 //tambem poderia acessar pelo ponteiro:
 p = &obj_b;
 p->mostra_nome();
 p = &obj_d;
 p->mostra_nome();
 //para acessar os componentes proprios de "derivada"
 //se necessita de um ponteiro especifico
 pd = &obj_d;
 pd->coloca_telefone("2245678");
 pd->mostra_nome();
 pd->mostra_telefone();
}
```

No exemplo, o ponteiro p está definido como apontador para a classe base. No entanto, ele pode apontar para objetos da classe derivada e servir para acessar os elementos da classe derivada que pertencem à classe base. Um ponteiro para a classe base não pode acessar elementos específicos da classe derivada. Por esta razão, para acessar o método mostra_telefone() foi necessário um ponteiro específico para a classe derivada (o ponteiro pd).

Exercícios:

1) O fonte "exer13.cpp" contém o exemplo acima. Compile e analise-o com auxílio do depurador.
2) Acrescente, na classe "base" do fonte "exer13.cpp", um campo para guardar o endereço da pessoa. Crie também operações sobre este campo. Depois crie uma segunda classe derivada, que herda as informações das classes "base" e "derivada", contendo campo e operações para número de fax. Na main(), faça manipulações com ponteiros sobre esta hierarquia de classes.

17. Funções Virtuais:

Uma função **virtual** é declarada como **virtual** em uma classe base e é redefinida em uma ou mais classes derivadas. Quando se acessa um objeto através de um ponteiro, em tempo de execução, é resolvido qual função efetivamente será executada, através do tipo apontado. Assim, quando se aponta para diferentes objetos, diferentes versões da função **virtual** poderão ser executadas. Isto é chamado "polimorfismo em tempo de execução" (equivalente ao obtido com apontadores para funções em C).

Para se declarar uma função como **virtual** em uma classe base, se precede sua declaração com a palavra reservada "**virtual**". Quando esta for redefinida na classe derivada não é necessário redefinir a palavra "**virtual**", embora isto não seja um erro. O exemplo a seguir mostra o uso de funções virtuais.

```
// Uso de funcoes virtuais:
class Base
      {
       public:
       virtual void who()
         {
          cout << "Classe Base\n";
         }
       };
//Uma classe derivada de Base:
class prim_d : public Base
      {
       public:
        void who()
         {
          cout << "Classe derivada primeira\n";
         }
       };
//Uma outra classe derivada de Base:
class sec_d : public Base
      {
       public:
        void who()
         {
          cout << "Classe derivada segunda\n";
         }
       };
void main()
{
 Base base, *p;
 prim_d prim, p1;
 sec_d  sec;
 p = &base;
 p->who();
 p = &prim;
 p->who();
 p = &sec;
 p->who();
}
```

Exercício:

1) O fonte "exer14.cpp" contém o código do exemplo anterior. Compile e execute o programa. Elimine a palavra reservada "**virtual**" na declaração da classe base. Compile e execute novamente o programa e veja o que acontece.
2) Desfaça a alteração proposta no exercício 1 e retire agora a declação de who() na classe sec_d. Compile e execute o programa para ver o resultado.

18. Funções virtuais puras - Classes abstratas

Numa implementação utilizando orientação a objetos, o ideal é que a classe base e todo o conjunto de classes derivadas possuam a mesma interface, ou seja, o mesmo conjunto de funções. Em muitos casos é comum que uma função não faça sentido para a classe base, mas apenas para as derivadas. Neste caso há duas soluções: o primeiro é definir, na classe base, uma função **virtual** que não possui processamento, ou que dá uma mensagem avisando que não está implementada. A segunda solução é definir uma função **virtual** pura, que é uma definição especial que obriga as derivadas a redefinir a função.

Uma função **virtual** pura é um tipo especial de função que se declara na classe base, e que não tem uma implementação nesta classe base. Como não há definição dela, cada derivada terá que definir sua própria versão: não podem simplesmente utilizar a versão que está na classe base. Para declarar uma função **virtual** pura, se usa o seguinte formato:

virtual *tipo nome-da-função(lista-de-parâmetros) = 0;*

Quando se declara uma função **virtual** pura, se força que cada derivada defina a sua própria implementação. Se uma classe não fizer isto, o compilador C++ indicará um erro.

O exemplo abaixo mostra o uso de uma função **virtual** pura:

```
//Exemplo de funcao virtual pura
class figura
      {
       protected:
        double x,y;
       public:
        void fornece_dim(double i, double j=0)
         {
          x = i;
          y = j;
         }
        //funcao virtual pura
        virtual void mostra_area()=0;
        };
class triangulo:public figura
    {
     public:
      void mostra_area()
       {
    printf("Triangulo de altura %d e base %d possui area de %f\n",x,y,x*0.5*y);
       }
    };
class retangulo:public figura
      {
       public:
        void mostra_area()
         {
      printf("Retangulo de dimensoes %d e %d tem area de %d\n",x,y,x*y);
          }
       };
class circulo:public figura
      {
       public:
        void mostra_area()
         {
            printf("Circulo de raio %d possui area de %f",x,3.14*x*x);
         }
       };
void main()
{
 figura *p; //ponteiro para o tipo base
 triangulo t;
```

```
 retangulo r;
 circulo c;
 p = &t;
 p->fornece_dim(10, 2.3);
 p = &r;
 p->fornece_dim(5, 5);
 p = &c;
 p->fornece_dim(2);
 t.mostra_area();
 r.mostra_area();
 p->mostra_area();
}
```

Se uma classe possui ao menos uma função **virtual** pura, então ela é dita uma classe abstrata. As classes abstratas possuem uma característica importante: não podem existir objetos desta classe. Por isto, uma classe abstrata deve ser usada apenas como base de herança para as outras classes. A razão para não existir objetos da classe abstrata é o fato de que uma ou mais de suas funções não possui definição. Entretanto, se podem declarar ponteiros para a classe abstrata, já que é necessário para suportar o polimorfismo em tempo de execução.

Exercício:

1) O fonte "exer15.cpp" contém o código do exemplo anterior. Tente criar um objeto f, da classe "figura". Veja a mensagem de erro do compilador.
2) Na classe "circulo", troque o nome da função "mostra_area()" para "mostra_area2()". Veja a mensagem de erro do compilador.
3) Inclua uma nova classe chamada "paralelepipedo". Altere a classe base para suportar figuras tridimensionais.
4) O fonte "exer16.cpp" contém um programa incompleto que demostra um dos usos das classes virtuais: a criação de estruturas de dados que armazenam objetos de tipos variados com funções semanticamente iguais. O programa possui um vetor de ponteiros para a classe "figura" de "NFIG" posições. Para cada uma das 5 posições o programa deve solicitar ao usuário um tipo de figura e criar um objeto deste tipo, dinamicamente, armazenando seu endereço na posição correspondente do vetor. O programa deve então solicitar os dados sobre todas as figuras e, posteriormente, imprimir as áreas de cada uma. Analise as classes já criadas e complete o "main" de maneira que o programa execute seu objetivo.
5) O fonte "exer17.cpp" contém uma estrutura de classes que permite a criação de uma lista encadeada capaz de armazenar objetos de classes diferentes. Analise o fonte e responda:
   a) Que característica em comum deverão ter todas as classes de objetos que poderão ser armazenadas nesta lista?
   b) Como se deve proceder para criar a lista?

19. Sobrecarga de operadores:

Em C++, o sentido dos operadores (+, -, =, = =, /, *, >>, <<,...) podem ser extendidos pelo programador, criando operadores próprios para as classes por ele construídas. Isto é chamado de "sobrecarga de operadores". Quando um operador é sobrecarregado, nada é perdido do seu significado original. Apenas ocorre que ele ganha um novo sentido quando é operado com objetos da classe onde foi sobrecarregado.

O formato geral da sobrecarga é:

```
tipo nome_da_classe::operator#(lista_de_argumentos)
```

O símbolo # deve ser substituído pelo operador que se deseja sobrecarregar.

Quando se sobrecarrega um operador binário, apenas o objeto à direita do operador é passado como parâmetro. O objeto à esquerda é passado implicitamente, através do ponteiro **this**.

```
//Exemplo de sobrecarga dos operadores de
//soma e atribuicao
class palavra
  {
    char str[100];
   public:
    palavra operator+(palavra p);
    palavra operator=(palavra p);
    void mostra();
    void assinala(char *string);
   };
//Sobrecarga do operador soma (+)
palavra palavra::operator+(palavra p)
{
 palavra temp;
 strcpy(temp.str, str); //copia a 1a. string
 strcat(temp.str, " "); //coloca espaco de separacao
 strcat(temp.str, p.str); //copia a 2a. string
 return(temp);
}
//Sobrecarga da atribuicao (=)
palavra palavra::operator=(palavra p)
{
 strcpy(str, p.str); //copia a string para a esquerda
 return(*this);
}
//Exibe o conteudo da string
void palavra::mostra()
{
   printf("%s\n",str);
}
//Copia um conteudo para o objeto
void palavra::assinala(char *string)
{
 strcpy(str, string);
}
void main()
{
 palavra a,b,c;
 a.assinala("Teste");
 b.assinala("de Sobrecarga");
 //faz a concatenacao de a e b
 c = a + b;
 c.mostra();
```

```
 a.assinala("Valor novo");
 //faz a atribuicao da string em a para b e c
 c = b = a;
 c.mostra();
 b.mostra();
}
```

Exercícios:

1) Porque a rotina "**operator**+", do exemplo anterior, se utiliza da variável auxiliar "temp" e a rotina "**operator**=" não precisa? O que aconteceria se implementássemos "**operator**+" sem o uso da variável auxiliar?
2) Porque a rotina "**operator**=" precisa retornar um parâmetro do tipo "palavra"? O que aconteceria se não retornasse?
3) O fonte "exer19.cpp" contém o exemplo anterior. Acrescente a sobrecarga do operador menos ("-"). Ele deve eliminar da string à esquerda todas as letras que estão na string a direita.

   Exemplo:

   "aaeiouu" - "ao" = "eiuu".
4) O fonte "exer20.cpp" possui a estrutura da classe "VETOR". Implemente a classe e faça um programa que teste o uso dos operadores propostos.

19.1 - Sobrecarga de ++ e -- :

Quando se sobrecarrega um operador unário com ++ e --, não existe passagem explícita de parâmetros, já que o único objeto utilizado é fornecido em **this**. Com ++ e --, porém, surge um problema: é que estes operadores atuam de forma diferente se forem utilizados de forma pós-fixada ou pré-fixada (x=a++ é diferente de x=++a). Para resolver a ambigüidade, sempre que o operador pós-fixado é utilizado, o código gerado pelo compilador passa como parâmetro um número inteiro, que não possui nenhuma finalidade, apenas resolve a ambigüidade. Veja o exemplo:

```
//Exemplo que mostra a sobrecarga de operadores
//unarios (++ e --)
#include <iostream.h>
#include <string.h>
#include <ctype.h>
class palavra
  {
    char str[100];
   public:
    palavra operator++(int p); /* posfixado */
    palavra operator++(); /* prefixado */
    palavra operator--(int p); /* posfixado */
    palavra operator--(); /* prefixado */
    void mostra();
    void assinala(char *string);
   };
//Sobrecarga do operador incremento (++) - posfixado
//palavra palavra::operator++(int p)
{
 palavra temp;
 int i;
 strcpy(temp.str, str); //salva o valor antigo
 for(i=0; str[i]!= '\0'; i++)
  str[i] = toupper(str[i]); //converte em maiuscula
 return(temp);
}
//Sobrecarga do operador incremento (++) - prefixado
//palavra palavra::operator++()
{
 int i;
 for(i=0; str[i]!= '\0'; i++)
```

```
  str[i] = toupper(str[i]); //converte em maiuscula
 return(*this);
}
//Sobrecarga do operador incremento (--) - posfixado
//palavra palavra::operator--(int p)
{
 palavra temp;
 int i;
 strcpy(temp.str, this->str); //salva o valor antigo
 for(i=0; str[i]!= '\0'; i++)
  str[i] = tolower(str[i]); //converte em minuscula
 return(temp);
}
//Sobrecarga do operador incremento (--) - prefixado
//palavra palavra::operator--()
{
 int i;
 for(i=0; str[i]!= '\0'; i++)
  str[i] = tolower(str[i]); //converte em minuscula
 return(*this);
}
//Exibe o conteudo da string
void palavra::mostra()
{
  printf("%s\n",str);
}
//Copia um conteudo para o objeto
void palavra::assinala(char *string)
{
 strcpy(str, string);
}
void main()
{
 palavra a,b;
 a.assinala("Teste");
 b.assinala("de Sobrecarga");
 a++;
 --b;
 a.mostra();
 b.mostra();
 a--;
 ++b;
 a.mostra();
 b.mostra();
}
```

Exercício:

1) Acrescente a classe "VETOR" (fonte "exer20.cpp") a sobrecarga dos operadores "++" e "--". As versões pré e pós fixadas devem ter o mesmo significado que no exemplo da classe "palavra" do exemplo anterior.

19.2 Sobrecarga de operadores usando funções "friend"

Quando se usa as funções "**operator**" como membros da classe sobre a qual o operador sobrecarregado tem influência, já notamos que o argumento a esquerda do operador é passado implicitamente (ponteiro "**this**"). Isso significa que a ativação da rotina depende do argumento que fica a esquerda do operador e que portanto este deve ser do tipo da classe em questão.

Existem situações, porém, que podemos querer criar operadores que possuam operandos de classes diferentes. A classe "VETOR" (fonte "exer20.cpp") é um exemplo típico onde pode-se criar um operador que some um escalar em um vetor. A implementação de tal operador poderia ser:

```
VETOR VETOR::operator+(float a)
  {
    VETOR temp;
    temp.x = x+a;
    temp.y = y+a;
    temp.z = z+a;
    return(temp);
  }
```

Apartir dessa rotina é possível escrever expressões do tipo:

```
VETOR a,b;
a = b+10.2;
```

porém, não é possível escrever: "a = 10.2+b;" porque a ativação da rotina se dá pelo elemento a esquerda do operador.

Para contornar esse problema podemos criar as funções "**operator**" como "friends". Como uma função "**friend**" não pertence efetivamente a classe, ela não recebe o ponteiro "**this**". Por essa razão a função deve ter dois argumentos, um para cada operando. Dessa forma pode-se criar versões onde argumentos de tipos diferentes aparecem de forma variada em ambos os lados dos operadores.

Ex:

```
class VETOR
  {
     ...
    friend VETOR operator+(VETOR op1,float op2);
    friend VETOR operator-(float op1,VETOR op2);
    ...
  };
VETOR operator+(VETOR op1,float op2)
  {
    VETOR temp;
    temp.x = op1.x+op2;
    temp.y = op1.y+op2;
    temp.z = op1.z+op2;
    return(temp);
  }
VETOR operator+(float op1,VETOR op2)
  {
    VETOR temp;
    temp.x = op1+op2.x;
    temp.y = op1+op2.y;
    temp.z = op1+op2.z;
    return(temp);
  }
```

Exercício:

1) Acrescente a classe "VETOR" (fonte "exer20.cpp") rotinas que sobrecarreguem os operadores de soma e subtração para as operações correspondentes de soma e subtração de escalares inteiros e reais com vetores.

20. Sobrecarregando os operadores "new" e "delete"

O operadores "**new**" e "**delete**" também podem ser sobrecarregados assim como os demais operadores. A sobrecarga deste tipo de operador pode ser necessária quando, por exemplo, deseja-se alterar o algoritmo de busca por memória livre (para usar memória **virtual**, otimizar a segmentação, etc).

O modelo de sobrecarga para "**new**" e "**delete**" é mostrado a seguir:

```
void *operator new(size_t tamanho)
  {
      //realiza alocação
      return( <ponteiro para memória alocada> );
  }
void *operator delete(void *p)
  {
      // libera memória apontada por *p
  }
```

Para sobrecarregar os operadores "**new**" e "**delete**" relativos a uma classe, simplesmente torne as funções "**operator**" membros de uma classe.

Ex:

```
void * VETOR::operator new(size_t tamanho)
  {
      return( malloc(size) );
  }
void VETOR::operator delete(void *p)
  {
      free(p);
  }
```

Exercício:

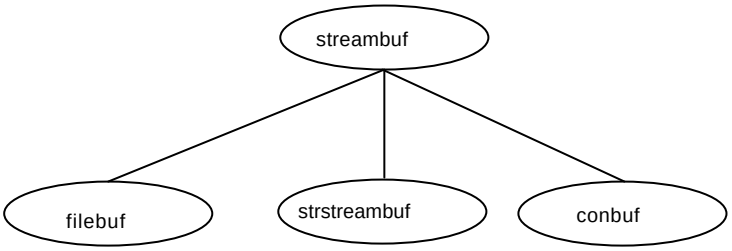
1) Defina uma classe MEMORIA que possui um vetor do tipo **char** com 32K posições e funções sobrecarregadas para "**new**" e "**delete**" que alocam área sobre este vetor.

21. Usando as Streams do C++

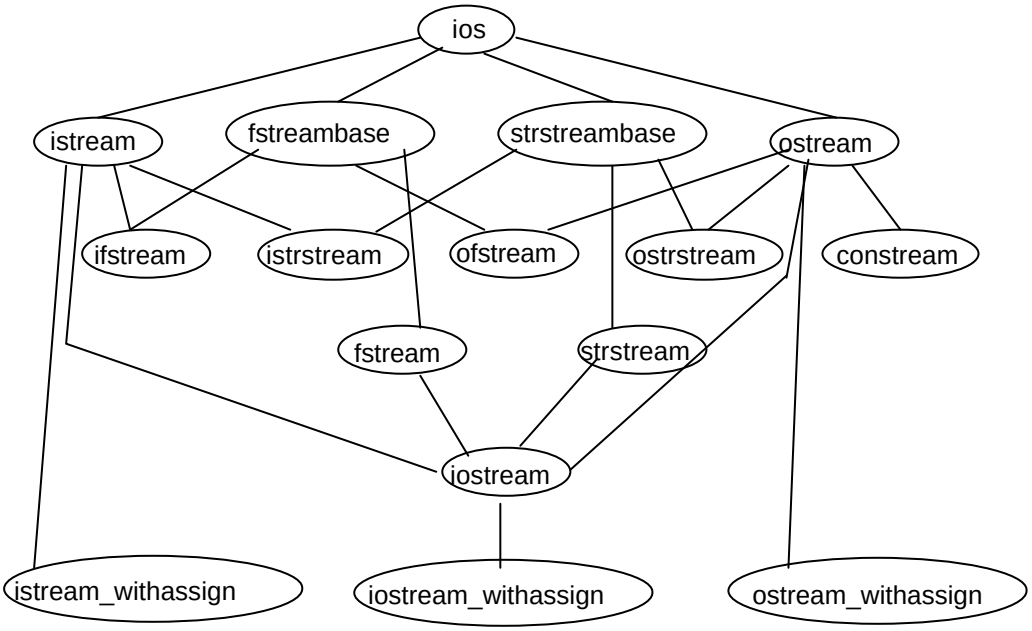
Uma stream é uma abstração que se refere a qualquer fluxo de dados de uma fonte a um destino. Em geral se usa as expressões "extrair" e "inserir" respectivamente para nos referirmos as operações de receber caracteres de uma fonte e armazenar caracteres em um destino. São providas classes que suportam saída pelo console (constrea.h), buffers de memória (iostream.h), arquivos (fstream.h) e strings (strstrea.h) como fontes, destinos ou ambos.

A biblioteca "iostream" possui duas familias paralelas de classes: aquelas derivadas de "streambuf" e aquelas derivadas de "ios". Ambas são classes de baixo nivel. Todas classes "stream"são derivadas de uma das duas. A comunicação entre ambas é feita por ponteiros.

A classe "streambuf" provê a interface com dispositivos físicos. Provê métodos gerais de para tratamento de buffers quando não há necessidade de formatação. "streambuf" é usada por outras partes da biblioteca "iostream" através da qual é possível derivar suas próprias classes.



A classe "ios" (e consequentemente todas as suas derivadas) contém um ponteiro para "streambuf". Ela provê entrada e saída formatada com checagem de erros.



Um programa "C++" começa com 4 streams pré-definidos abertos, declarados como objetos da classe "withassign" como segue:

```
extern istream_withassign cin;    // corresponde a stdin
extern ostream_withassign cout;   // corresponde a stdout
extern ostream_withassign cerr;   // corresponde a stderr
extern ostream_withassign clog;   // um cerr buferizado
```

21.1 Saída de dados usando streams

Um stream de saída é acompanhado do operador de direcionamento de fluxo ou operador de inserção "<<". O operador original de deslocamento de bits é sobrecarregado para as operações de saída. O operando esquerdo desse operador é um objeto do tipo "ostream" e o operador direito pode ser de qualquer tipo de dados para o qual o operador "<<" tiver sido origunalmente definido (todos os tipos de dados standart) ou posteriormente sobre carregado.

Ex:

```
cout << "Hello\n";
```

O operador "<<" associa da esquerda para a direita e e retorna uma referência para um objeto "ostream" o que permite seu uso em cascata. Ex:

```
cout << "O numero e " << x << "\n";
```

Os tipos fundamentais diretamente suportados são: **char**, short, **int**, long, **char*** (tratado como string), **float**, **double**, long **double** e **void***. O formato de saída desses tipos é o mesmo definido para um "printf", a menos que isso seja alterado através das "flags" de formatação. O tratamento para "**void***" permite a impressão de endereços de memória. Ex:

```
int i;
cout  << &i;
```

A formatação da saída é determinada por várias "flags" de formatação definidas na classe "ios" ou através de manipuladores. Manipuladores recebem uma referencia para um stream como argumento e retornam uma referência para este mesmo stream. Os manipuladores podem ser inseridos em uma cadeia de inserções ou extrações para alterar o estado dos streams.

Ex:

```
#include <iostream.h>
#include <iomanip.h>
int main(void)
  {
    it i = 6789, j = 1234, k = 10;

    cout << setw(6) << i << j << i << k << j;
    cout << '\n';
    cout << setw(6) << i << setw(6) << j << setw(6) << k;
    return(0);
  }
```

Saída esperada:

```
678912346789101234
6789    1234     10
```

Principais manipuladores:

dec	seleciona base decimal
hex	seleciona base hexadecimal
oct	seleciona base octal
ws	elimina os caracteres brancos
endl	insere um "\n" e limpa o stream
ends	insere um "\0" em um string
setbase (**int** n)	seleciona a base de conversão para "n" (0, 8, 10 ou 16).
	O significa o default: decimal na saída.
resetiosflags(long f)	limpa os bits de formato especificados por "f"
setiosflags(long f)	seleciona os bits de formatação especificados por "f".
setprecision(**int** n)	seleciona a precisão do ponto flutuante
setw(**int** n)	seleciona a largura do campo para "n".

Principais flags:

left	ajuste dos campos pela esquerda
right	ajuste dos campos pela direita
dec,oct,hex	selcionam base de conversão

showbase	indica a base na saída
showpoint	mostra o ponto decimal (com ponto flutuante)
uppercase	saída hexadecimal em maiúsculas
showpos	mostra o sinal "+" para inteiros positivos
scientific	usa notação científica com ponto flutuante
fixed	usa ponto decimal fixo com números ponto flutuante

O caracter "default" para preenchimento do alinhamento dos campos é o espaço em branco. Isso pode ser alterado, porém, pela função membro "fill".

Ex:

```
int i = 123;
cout.fill('*');
cout << setw(6) << i << '\n';
```

21.2 Entrada de dados usando streams

A entrada de dados usnado stream é similar a saída, porém, usa o operador de deslocamento de bits ">>" sobrecarregado. O operador esquerdo de ">>" é um objeto da classe "istream" e o direito é qualquer um dos tipos definidos para o mesmo.

Os tipos de dados numéricos são lidos ignorando os espaços em branco. Uma leitura para um elemento do tipoo "**char**" implica na leitura do primeiro caracter diferente de branco. Para leitura de strings são lidos todos os caracteres delimitados entre dois espaços em branco (ou pelo início do campo e um "\n"). Para evitar o overflow do string pode-se usar a função membro "width".

Ex:

```
char array[SIZE];
cin.width(sizeof(array));
cin >> array;
```

Exercícios

1) Substitua todas as ocorrências de "printf" e "scanf" do fonte "exer16.cpp" pelos objetos "cin" e "cout".
2) Faça um programa em "C++" que lê uma série de informações sobre os produtos de uma empresa, armazene essas informações em um vetor e depois imprima uma tabela formatada que exiba esses dados.

   Informações a serem lidas sobre cada produto:

código	- **char** (6)
descrição	- **char** (20)
quantidade	- **int** (4)
preço	- **float**(9.2)

21.3 Sobrecarregando os operadores de entrada e saída

Os operadores ">>" e "<<" quando usados com "cin" e "cout" são chamados de introdutores e extratores pelo fato de introduzirem e retirarem dados de um fluxo de E/S. Estes operadores podem ser sobrecarregados de maneira que os mesmos passem a suportar também os tipos de dados definidos no programa.

Exemplo de sobrecarga dos "introdutores" e "extratores" para a classe "VETOR" (fonte "exer20.cpp") :

```
class VETOR
  {
    ...
    friend ostream operator<<( ostream &stream, VETOR obj)
    friend istream  operator>>( istream  &stream, VETOR &obj);
    ...
  };
ostream operator<<( ostream &stream,VETOR obj)
  {
```

```
    stream << obj.x << ",";
    stream << obj.y << ",";
    stream << obj.z << "\n";
    return(stream);
  }

istream  operator>>( istream  &stream, VETOR &obj)
  {
    cout << "Informe os valores para X, Y e Z: ";
    stream >> x >> y >> z;
    return( stream );
  }
```

As funções "**operator**" para introdutores e extratores não podem ser membro das classes (por exemplo "VETOR") porque a ativação se daria pelo elemento a esquerda do operador (que certamente não seria um "VETOR"). Por esta razão elas devem ser declaradas como "**friend**".

Exercício:

1) Sobrecarregue os operadores "introdutores" e "extratores" para a classe "FUNCIONARIOS"(fonte "exer8.cpp").

21.4 Manipulação simples de arquivos

A classe "ofstream" herda as operações de inserção de "ostream" enquanto que "ifstream" herda as operações de extração de "istream". As classes stream para manipulação de arquivos provem construtores e funções membro para a criação de arquivos e gravação e leitura de dados.

Ex:

```
#include <fstream.h>
void main()
  {
    char ch;
    ifstream f1("FILE.IN");
    ofstream f2("FILE>OUT");
    if (!f1) cerr << "Nao pode abrir FILE.IN para leitura\n";
    if (!f2) cerr << "Nao pode abrir FILE.OUT para gravação\n";
    while(f2 && f1.get(ch))
        f2.put(ch);
    return(0);
  }
```

As classes "ifstream" e "ofstream" possuem construtoras que facilitam a abertura de arquivos nos modos defautl. É possível entretanto utilizar a função membro "open"para uma abertura mais elaborada. A função membro "close" pode ser usada para o fechamento do arquivo e a função "eof" para detecção de fim de arquivo.

Por default os arquivos são abertos em modo texto.

Ex:

```
ofstream ofile;
...
ofile.open("pgtos",ios::binary|ios::app);
....
ofile.close();
```

O parâmetro "flags" da função "open" é opcional. As possibilidades são as seguintes:

ios**::**app - acrescenta dados no fim do arquivo

ios**::**in - abre para leitura

ios**::**out - abre para gravação

ios**::**binary - abre em modo binário

ios**::**nocreate - se o arquivo existe, open falha

ios**::**noreplace - se o arquivo existe open falha a menos que tenha a opção app.

Exercício:

1) Faça um programa em "C++" que cria um arquivo texto com uma lista de nomes. Obs: use o operador ">>" para gravar os nomes.
2) Faça um programa em "C++" que simula o comando "type" do DOS.

21.5 Streams de processamento de strings

As funções definidas em "strstrea.h" possuem funcionalidade semelhante a dos comandos "sscanf" e "sprintf".

Exemplo:

```
#include <fstream.h>
#include <strstrea.h>
#include <iomanip.h>
#include <string.h>
int main(int argc,char **argv)
  {
    int id;
    float amount;
    char description[41];
    ifstream inf(argv[1]);
    if (inf)
      {
        char inbuf[81];
        int lineno = 0;
        while(inf.getline(infbuf,81))
          {
         isstream ins(inbuf,strlen(inbuf));
         ins >> id >> amount >> ws;
         ins.getline(description,41);
         cout << ++lineno << ": "
              << setprecision(2) << amount << '\t'
               << description << "\n";
        }
    }
  }
```

22. Templates

Templates são um recurso que permite que se montem "esqueletos" de funções ou classes onde a definição dos tipos de dados fica postergada para o momento do uso da classe ou função. Este tipo de recurso é extremamente interessante em uma linguagem "tipada" como "C++" onde os tipos de dados sempre devem ser declarados nos argumentos de funções e atributos de classes limitando a definição de estruturas genéricas.

22.1 Funções "templates"

Permite a criação de funções onde o tipo dos argumentos é definido na hora da chamada da função. Desta forma, com uma única declaração, é possível definir uma familia de funções sobrecarregadas com a vantagem que as mesmas só serão efetivamente "geradas" no caso de serem utilizadas.

Exemplo:

```
template <class T>
void f1(T a,T b)
  {
    cout << "Soma: "<< (a+b) <<"\n";
  }
void main()
  {
    int a,b;
    float d,e;
    clrscr();
    a = 10;  b = 20;
    d = 50.3; e = 45.8;
    f1(a,b);
    f1(d,e);
    getch();
  }
```

22.2 Classes "templates"

Uma "classe **template**" visa a definição de uma familia de classes que possuem estrutura semelhante. As maiores diferenças entre o uso de uma "classe **template**" e uma classe comum aparecem na implementação das rotinas fora do corpo da classe e na criação de objetos:

Exemplo:

```
template <class T>
class List
{
  protected:
  T *v;
  int size;

  public:
  List(int);
  void Set(T v,int n);
  T Get(int n);
};
template <class T> List<T>::List(int n)
{
  v = new T[n];
  size = n;
}
template <class T> void List<T>::Set(T val,int n)
  {
    if ((n>0)&&(n<size))
      v[n] = val;
```

```
  }
template <class T> T List<T>::Get(int n)
  {
    if ((n>0)&&(n<size))
      return(v[n]);
    return(v[0]);
  }
void main()
  {
    List <int>vet(10);
    int r;
    for(r=0; r<10; r++)
      vet.Set(r*10,r);
    for(r=0; r<10; r++)
      cout << vet.Get(r) << "\n";
    getch();
  }
```

Exercícios:

1) O fonte "exer23.cpp" possui a declaração de uma classe "**template**" chamada "list" que implementa um vetor com controle de acesso fora de faixa. Faça um programa em "C++" que use esta classe para definir três vetores distintos:

   - Um vetor de inteiros de 10 posições;

   - Um vetor de **float** de 5 posições

   - Um vetor de "TNOME" de 3 posições, onde TNOME é uma classe que possui um atributo para guardar um nome de pessoa com até 40 caracteres.

   O programa deve ler valores para os vetores e imprimir estes valores.

22.3 Derivando "classes templates"

A derivação de uma classe **template** segue as mesmas regras de derivação de uma classe comum. Devem ser tomados alguns cuidados apenas em relação a sintaxe:

Exemplo:

```
template <class T>
class List
{
  public:
  T *v;
  int size;
  List(int);
  void Set(T v,int n);
  T Get(int n);
};
template <class T> List<T>::List(int n)
{
  v = new T[n];
  size = n;
}
template <class T> void List<T>::Set(T val,int n)
  {
    if ((n>0)&&(n<size))
      v[n] = val;
  }
template <class T> T List<T>::Get(int n)
  {
    if ((n>0)&&(n<size))
      return(v[n]);
```

```
    return(v[0]);
  }
template <class X>
class derivada : public List<X>
  {
    public:
    derivada(int n);
    void imp(void);
  };
template <class X> void derivada<X>::derivada(int n)
                              :List<X>(n)
  {
  }
template <class X> void derivada<X>::imp()
  {
    for(int r=0;r<size; r++)
      cout << v[r] << "\n";
  }
void main()
  {
    List <int>vet(10);
    derivada <float>vetf(20);
    int r;
    float f;
    for(r=0; r<10; r++)
      vet.Set(r*10,r);
    for(r=0; r<10; r++)
      cout << vet.Get(r) << "\n";
    for(r=0,f=0.0; r<20; r++,f+=34.0)
      vetf.Set(f,r);
    vetf.imp();
    getch();
  }
```

Exercício:

1) Derive uma classe **template** da classe "List"(definida em exer23.cpp) que seja capaz de informar o maior valor que foi armazenado no vetor até o momento da consulta,

23. Trabalhando com números em BCD

O BorlandC++ define a classe BCD que trabalha com números inteiros e reais em formato BCD. A classe BCD possui as seguintes funções construtoras:

```
BCD(int n);
BCD(double n);
BCD(double n,int digitos); //digitos = nro. de casas após a vírgula.
```

A classe BCD automaticamente sobrecarrega todos os operadores relacionais, aritméticos bem como a maioria das funções matemáticas. A classe BCD esta definida em <BCD.H>.

Exercício:

1) O fonte "exer22.cpp" possui um exemplo de uso de números em BCD. Compile-o, execute-o e veja os resultados. Altere agora as variáveis de "bcd" para "**double**" e veja as diferenças de precisão.