Curso de Programação – III

Introdução a POO

Linguagem de Programação C++

Autores:
Renato Borges
André Clínio

Conteúdo

Capítulo 1

Qualidade do software

O principal objetivo da Engenharia de Software é contribuir para a produção de programas de qualidade. Esta qualidade, porém, não uma idéia simples; mas sim como um conjunto de noções e fatores.

É desejável que os programas produzidos sejam rápidos, confiáveis, modulares, estruturados, modularizados, etc. Esses qualificadores descrevem dois tipos de qualidade:

- De um lado, considera-se aspectos como eficiência, facilidade de uso, extensibilidade, etc. Estes elementos podem ser detectados pelos usuários do sistema. A esses fatores atribui-se a qualidade externa do sofware.
- Por outro lado, existe um conjunto de fatores do programa que só profissionais de computação podem detectar. Por exemplo, legibilidade, modularidade, etc. A esses fatores atribui-se a qualidade interna do sofware.

Na realidade, qualidade externa do programa em questão é que importa quando de sua utilização. No entanto, os elementos de qualidade interna são a chave para a conquista da qualidade externa. Alguns fatores de qualidade externa são apresentados na próxima seção. A seção posterior trata da qualidade interna, analisando como a qualidade externa pode ser atingida a partir desta.

Fatores de qualidade externa

Corretude

É a condição do programa de produzir respostas adequadas e corretas cumprindo rigorosamente suas tarefas de acordo com sua especificação.

Obviamente, é uma qualidade primordial. Se um sistema não faz o que ele foi proposto a fazer, então qualquer outra questão torna-se irrelevante.

Robustez

É a capacidade do programa de funcionar mesmo sob condições anormais.

A robustez do programa diz respeito ao que acontece quando do aparecimento de situações anômalas. Isto é diferente de corretude, que define como o programa se comporta dentro de sua especificação. Na robustez, o programa deve saber encarar situações que não foram previstas sem efeitos colaterais catastróficos. Neste sentido, o termo confiabilidade é muito utilizado para robustez; porém denota um conceito mais amplo que é mais bem interpretado como que englobando os conceitos de corretude e robustez.

Extensibilidade

É definida como a facilidade com que o programa pode ser adaptado para mudanças em sua especificação.

Neste aspecto, dois princípios são essenciais:

- Simplicidade de design: uma arquitetura simples sempre é mais simples de ser adaptada ou modificada
- Descentralização: quanto mais autônomos são os módulos de uma arquitetura, maior será a probabilidade de que uma alteração implicará na manutenção de um ou poucos módulos.

Capacidade de reuso

É a capacidade do programa de ser reutilizado, totalmente ou em partes, para novas aplicações.

A necessidade de reutilização vem da observação de que muitos elementos dos sistemas seguem padrões específicos. Neste sentido, é possível explorar este aspecto e evitar que a cada sistema produzido os programadores fiquem "reinventando soluções" de problemas já resolvidos anteriormente.

Compatibilidade

É a facilidade com que o programa pode ser combinado com outros.

Esta é uma qualidade importante pois os programas não são desenvolvidos *stand-alone*. Normalmente, existe uma interação entre diversos programas onde o resultado de um determinado sistema é utilizado como entrada para outro.

Outros aspectos

Os aspectos resumidos acima são aqueles que podem ser beneficiados com a boa utilização das técnicas de orientação por objetos. No entanto, existem outros aspectos que não podem ser esquecidos:

- Eficiência: É o bom aproveitamento dos recursos computacionais como processadores, memória, dispositivos de comunicação, etc. Apesar de não explicitamente citado anteriormente, este é um requisito essencial para qualquer produto. A linguagem C++ em particular, por ser tão eficiente quanto C, permite o desenvolvimento de sistemas eficientes.
- Portabilidade: É a facilidade com que um programa pode ser transferido de uma plataforma (*hardware*, sistema operacional, etc.). Este fator depende muito da base sobre a qual o sistema é desenvolvido. A nível de linguagem, C++ satisfaz este fator por ser uma linguagem padronizada com implementações nas mais diversas plataformas.
- Facildade de uso: É a facilidade de aprendizagem de como o programa funciona, sua operação, etc.

Modularidade - qualidade interna

Alguns fatores apresentados na seção anterior podem ser beneficiados com o uso de orientação por objetos. São eles: corretude, robustez, extensibilidade, reuso e compatibilidade. Estes refletem as mais sérias questões para o desenvolvimento de programas. A medida que se olha para estes cinco aspectos, dois subgrupos surgem:

- Extensibilidade, reuso e compatibilidade demandam arquiteturas que sejam flexíveis, design descentralizado e a construção de módulos coerentes conectados por interfaces bem definidas.
- Corretude e robustez são favorecidos por técnicas que suportam o desenvolvimento de sistemas baseados em uma especificação precisa de requisitos e limitações.

Da necessidade da construção de arquiteturas de programas que sejam flexíveis, surge a questão de tornar os programas mais modulares.

Uma definição precisa para modularidade é dificilmente encontrada. No entanto, esta técnica não traz benefícios se os módulos não forem autonômos, coerentes e organizados em uma estrutura robusta. Será utilizado no decorrer deste texto um conjunto de cinco critérios e cinco princípios.

Critérios para modularidade

Seguem abaixo cinco critérios que ajudam a avaliar/construir a modularidade do *design*:

Decomposição

O critério de decomposição modular é alcançado quando o modelo de design ajuda a decomposição do problema em diversos outros subproblemas cujas soluções podem ser atingidas separadamente.

O método deve ajudar a reduzir a aparente complexidade de problema inicial pela sua decomposição em um conjunto de problemas menores conectados por uma estrutura simples. De modo geral, o processo é repetitivo: os subproblemas são também divididos em problemas menores e assim sucessivamente.

Uma exemplificação deste tipo de *design* é o chamado método *top-down*. Este método dirige os desenvolvedores a começar com uma visão mais abstrata do funcionamento do sistema. Esta visão abstrata vai sendo refinada como um conjunto de passos sucessivos menores e assim por diante até que seus elementos estejam em um nível que permita sua implementação. Este processo pode ser modelado como uma árvore.

Composição

O método que satisfaz o critério de composição favorece a produção de elementos de programas que podem ser livremente combinados com os outros de forma a produzir novos sistemas, possivelmente em um ambiente bem diferente daquele em que cada um destes elementos foi criado.

Enquanto que a decomposição se concentra na divisão do software em elementos menores a partir da especificação, a composição se estabelece no sentido oposto: agregando elementos de programas que podem ser aplicados para construção de novos sistemas.

A composição é diretamente relacionada com a questão da reutilização: o objetivo é achar maneiras de desenvolver pedaços de programas que executam tarefas bem definidas e utilizáveis em outros contextos. Este contexto reflete um sonho antigo: transformar o processo de design de programas como elementos independentes onde programas são construídos pela combinação de elementos existentes.

Um exemplo deste tipo de abordagem é a construção de bibliotecas como conjuntos de elementos que podem ser utilizados em diversos programas (pacotes gráficos, bibliotecas numéricas, etc.).

Entendimento

Um método que satisfaz o critério de entendimento ajuda a produção de módulos que podem ser separadamente compreeendidos pelos desenvolvedores; no pior caso, o leitor deve ter atenção sobre poucos módulos vizinhos. Este critério é especialmente relevante quando se tem em vista o aspecto da manutenção.

Um contra-exemplo deste tipo de método é quando há dependência sequencial onde um conjunto de módulos é elaborado de modo que a execução dos mesmos seja feita em uma ordem determinada. Desta forma, os módulos não são entendidos de forma individual mas em conjunto com seus vizinhos.

Continuidade

Um método de design satisfaz a continuidade se uma pequena mudança na especificação do problema resulta em alterações em um único ou poucos módulos. Tal alteração não tem reflexos na arquitetura geral do sistema; isto é, no relacionamento intermodular.

Este critério reflete o problema de extensibilidade do sistema. A continuidade significa que eventuais mudanças devem afetar os módulos individualmente da estrutura do sistema e não a estrutura em si.

Um exemplo simples deste tipo de critério é a utilização de constantes representadas por nomes simbólicos definidos em um único local. Se o valor deve ser alterado, apenas a definição deve ser alterada.

Proteção

Um método de design satisfaz a proteção se este provê a arquitetura de isolamento quando da ocorrência de condições anômalas em tempo de execução. Ao aparecimento de situações anormais, seus efeitos ficam restritos àquele módulo ou pelo menos se propagar a poucos módulos vizinhos.

Os erros considerados neste critério são somente aqueles ocorridos em tempo de execução como falta de espaço em disco, falhas de hardware, etc. Não se considera, neste caso, a correção de erros, mas um aspecto importante para a modularidade: sua propagação.

Princípios de modularidade

Estabelecidos os critérios de modularidade, alguns princípios surgem e devem ser observados cuidadosamente para se obter modularidade. O primeiro princípio se relaciona com a notação e os outros se baseiam no modo de comunicação entre os módulos.

Seguem abaixo estes princípios:

Linguística modular

Este princípio expressa que o formalismo utilizado para expressar o design, programas, etc. deve suportar uma visão modular; isto é:

Módulos devem correspoder às unidades sintáticas da linguagem utilizada.

Onde a linguagem utilizada pode ser qualquer linguagem de programação, de design de sistemas, de especificação, etc.

Este princípio segue de diversos critérios mencionados anteriormente:

- Decomposição: Se pretende-se dividir o desenvolvimento do sistema em tarefas separadas, cada uma destas deve resultar em uma unidade sintática bem delimitada. (compiláveis separadamente no caso de linguagens de programação)
- Composição: Só pode-se combinar coisas como unidades bem delimitadas.
- Proteção: Somente pode haver controle do efeito de erros se os módulos estão bem delimitados.

Poucas interfaces

Este princípio restringe o número de canais de comunicação entre os módulos na arquitetura do sistema. Isto quer dizer que:

Cada módulo deve se comunicar o mínimo possível com outros.

A comunicação pode ocorrer das mais diversas formas. Módulos podem chamar funções de outros, compartilhar estruturas de dados, etc. Este princípio limita o número destes tipos de conexão.

Pequenas interfaces

Este princípio relaciona o tamanho das interfaces e não suas quantidades. Isto quer dizer que:

Se dois módulos possuem canal de comunicação, estes devem trocar o mínimo de informação possível; isto é, os canais da comunicação intermodular devem ser limitados.

Interfaces explícitas

Este é um princípio que vai mais adiante do que “poucas interfaces” e “pequenas interfaces”. Além de impor limitações no número de módulos que se comunicam e na quantidade de informações trocadas, há a imposição de que se explicite claramente esta comunicação. Isto é:

Quando da comunicação de dois módulos A e B, isto deve ser explícito no texto de A, B ou ambos.

Este princípio segue de diversos critérios mencionados anteriormente:

- Decomposição e composição: Se um elemento é formado pela composição ou decomposição de outros, as conexões devem ser bem claras entre eles.
- Entendimento: Como entender o funcionamento de um módulo A se seu comportamento é influenciado por outro módulo B de maneira não clara?

Ocultação de informação (information hiding)

A aplicação deste princípio assume que todo módulo é conhecido através de uma descrição oficial e explícita; ou melhor sua interface. Pela definição, isto requer que haja uma visão restrita do módulo.

Toda informação sobre um módulo deve ser oculta (privada) para outro a não ser que seja especificamente declarada pública.

A razão fundamental para este princípio é o critério de continuidade. Se um módulo precisa ser alterado, mas de maneira que só haja manutenção de sua parte não pública e não a interface; então os módulos que utilizam seus recursos (clientes) não precisam ser alterados. Além, quanto menor a interface, maiores as chances de que isto ocorra.

Apesar de não haver uma regra absoluta sobre o que deve ser mantido público ou não, a idéia geral é simples: a interface deve ser uma descrição do funcionamento do módulo. Tudo que for relacionado com sua implementação não deve ser público.

É importante ressaltar que, neste caso, este princípio não implica em proteção no sentido de restrições de segurança. Embora seja invisível, pode haver acesso às informações internas do módulo.

Calculadora RPN em C

Os primeiros exemplos serão feitos a partir de um programa escrito em C. Várias modificações serão feitas até que este se torne um programa C++. O programa é uma calculadora RPN (notação polonesa reversa), apresentada nesta seção.

Este tipo de calculadora utiliza uma pilha para armazenar os seus dados. Esta pilha está implementada em um módulo à parte. O header file etá listado abaixo:

```
#ifndef stack_h
#define stack_h

#define MAX 50

struct Stack {
  int top;
  int elems[MAX];
};

void push(struct Stack* s, int i);
int  pop(struct Stack* s);
int  empty(struct Stack* s);
struct Stack* createStack(void);

#endif
```

A implementação destas funções está no arquivo stack-c.c:

```
#include <stdlib.h>
#include "stack-c.h"

void push(struct Stack*s, int i) { s->elems[s->top++] = i; }
int  pop(struct Stack*s)         { return s->elems[--(s->top)]; }
int  empty(struct Stack*s)       { return s->top == 0; }

struct Stack* createStack(void)
{
   struct Stack* s = (struct Stack*)malloc(sizeof(struct Stack));
   s->top = 0;
   return s;
}
```

A calculadora propriamente dita utiliza estes arquivos:

```
#include <stdlib.h>
#include <stdio.h>
#include "stack-c.h"

/* dada uma pilha, esta função põe nos
   parâmetros n1 e n2 os valores do topo
   da pilha. Caso a pilha tenha menos de dois
   valores na pilha, um erro é retornado */
int getop(struct Stack* s, int* n1, int* n2)
{
   if (empty(s))
   {
      printf("empty stack!\n");
      return 0;
   }
   *n2 = pop(s);
   if (empty(s))
   {
      push(s, *n2);
      printf("two operands needed!\n");
      return 0;
   }
   *n1 = pop(s);
   return 1;
}
```

```
/* a função main fica em um loop
   lendo do teclado os comandos da calculadora.
   Se for um número, apenas empilha.
   Se for um operador, faz o calculo.
   Se for o caracter 'q', termina a execução.
   Após cada passo a pilha é mostrada. */
int main(void)
{
   struct Stack* s = createStack();
   while (1)
   {
      char str[31];
      int i;
      printf("> ");
      gets(str);
      if (sscanf(str, " %d", &i)==1) push(s, i);
      else
      {
         int n1, n2;
         char c;
         sscanf(str, "%c", &c);
         switch(c)
         {
            case '+':
               if (getop(s, &n1, &n2)) push(s, n1+n2);
               break;
            case '-':
               if (getop(s, &n1, &n2)) push(s, n1-n2);
               break;
            case '/':
               if (getop(s, &n1, &n2)) push(s, n1/n2);
               break;
            case '*':
               if (getop(s, &n1, &n2)) push(s, n1*n2);
               break;
            case 'q':
               return 0;
            default:
               printf("error\n");
         }
      }
      {
         int i;
         for (i=0; i<s->top; i++)
            printf("%d:%6d\n", i, s->elems[i]);
      }
   }
}
```

Tipos abstratos de dados

O conceito de tipo foi um passo importante no sentido de atingir uma linguagem capaz de suportar programação estruturada. Porém, até agora, não é possível usar uma metodologia na qual os programas sejam desenvolvidos por meio de decomposições de problemas baseadas no reconhecimento de abstrações. Para a abstração de dados adequada a este propósito não basta meramente classificar os objetos quanto a suas estruturas de representação; em vez disso, os objetos devem ser classificados de acordo com o seu comportamento esperado. Esse comportamento é expresso em termos das operações que fazem sentido sobre esses dados; estas operações são o único meio de criar, modificar e se ter acesso aos objetos.

Classes em C++

Uma classe em C++ é o elemento básico sobre o qual toda orientação por objetos está apoiada. Em primeira instância, uma classe é uma extensão de uma estrutura, que passa a ter não apenas dados, mas

também funções. A idéia é que tipos abstratos de dados não são definidos pela sua representação interna, e sim pelas operações sobre o tipo. Então não há nada mais natural do que incorporar estas operações no próprio tipo.Estas operações só fazem sentido quando associadas às suas representações.

No exemplo da calculadora, um candidato natural a se tornar uma classe é a pilha. Esta é uma estrutura bem definida; no seu header file estão tanto a sua representação (*struct Stack*) quanto as funções para a manipulação desta representação. Seguindo a idéia de classe, estas funções não deveriam ser globais, mas sim pertencerem à estrutura *Stack*. A função *empty* seria uma das que passariam para a estrutura. A implementação de *empty* pode ficar dentro da própria estrutura:

```
struct Stack {
   // ...
   int empty() { return top == 0; }
};
```

ou fora:

```
struct Stack {
   // ...
   int empty();
};

int Stack::empty(void) { return top == 0; }
```

No segundo caso a declaração fica no header file e a implementação no .c. A diferença entre as duas opções será explicada na seção sobre funções inline.

Com esta declaração, as funções são chamadas diretamente sobre a variável que contém a representação:

```
void main(void)
{
   struct Stack s;
   s.empty();
}
```

Repare que a função deixou de ter como parâmetro uma pilha. Isto era necessário porque a função estava isolada da representação. Agora não, a função faz parte da estrutura. Automaticamente todos os campos da estrutura passam a ser visíveis dentro da implementação da função, sem necessidade se especificar de qual pilha o campo *top* deve ser testado (caso da função *empty*). Isto já foi dito na chamada da função.

O paradigma da orientação a objetos

Esta seção apresenta cinco componentes chave do paradigma de orientação por objetos. As componentes são objeto, mensagem, classe, instância e método. Eis as definições:

- Objeto: é uma abstração encapsulada que tem um estado interno dado por uma lista de atributos cujos valores são únicos para o objeto. O objeto também conhece uma lista de mensagens que ele pode responder e sabe como responder cada uma. Encapsulamento e abstração são definidos mais à frente.
- Mensagem: é representada por um identificador que implica em uma ação a ser tomada pelo objeto que a recebe. Mensagens podem ser simples ou podem incluir parâmetros que afetam como o objeto vai responder à mensagem. A resposta também é influenciada pelo estado interno do objeto.
- Classe: é um modelo para a criação de um objeto. Inclui em sua descrição um nome para o tipo de objeto, uma lista de atributos (e seus tipos) e uma lista de mensagens com os métodos correspondentes que o objeto desta classe sabe responder.
- Instância: é um objeto que tem suas propriedades definidas na descrição da classe. As propriedades que são únicas para as instâncias são os valores dos atributos.
- Método: é uma lista de instruções que define como um objeto responde a uma mensagem em particular. Um método tipicamente consiste de expressões que enviam mais mensagens para objetos. Toda mensagem em uma classe tem um método correspondente.

Traduzindo estas definições para o C++, classes são estruturas, objetos são variáveis do tipo de alguma classe (instância de alguma classe), métodos são funções de classes e enviar uma mensagem para um objeto é chamar um método de um objeto.

Resumindo:

Objetos são instâncias de classes que respondem a mensagens de acordo com os métodos e atributos, descritos na classe.

Exercício 1 - Calculadora RPN com classes

Transformar a pilha utilizada pela calculadora em uma classe.

Capítulo 2

Descrição de recursos de C++ não relacionados às classes

Comentários

O primeiro recurso apresentado é simplesmente uma forma alternativa de comentar o código. Em C++, os caracteres // iniciam um comentário que termina no fim da linha na qual estão estes caracteres. Por exemplo:

```
int main(void)
{
   return -1; // retorna o valor -1 para o sistema operacional
}
```

Declaração de variáveis

Em C as variáveis só podem ser declaradas no início de um bloco. Se for necessário usar uma variável no meio de uma função existem duas soluções: voltar ao início da função para declarar a variável ou abrir um novo bloco, simplesmente para possibilitar uma nova declaração de variável.

C++ não tem esta limitação, e as variáveis podem ser declaradas em qualquer ponto do código:

```
void main(void)
{
   int a;
   a = 1;
   printf("%d\n", a);
   // ...
   char b[] = "teste";
   printf("%s\n", b);
}
```

O objetivo deste recurso é minimizar declarações de variáveis não inicializadas. Se a variável pode ser declarada em qualquer ponto, ela pode ser sempre inicializada na própria declaração. Um exemplo comum é o de variáveis usadas apenas para controle de loops:

```
for (int i=0; i<20; i++) // ...
```

A variável *i* está sendo criada dentro do for, que é o ponto onde se sabe o seu valor inicial. O escopo de uma variável é desde a sua declaração até o fim do bloco corrente (fecha chaves). Na linha acima a variável *i* continua existindo depois do for, até o fim do bloco.

É interessante notar que esta característica é diferente de se abrir um novo bloco a cada declaração. Uma tentativa de declarar duas variáveis no mesmo bloco com o mesmo nome causa um erro, o que não acontece se um novo bloco for aberto.

Declaração de tipos

Em C++, as declarações abaixo são equivalentes:

```
struct a {
   // ...
};
typedef struct a {
   // ...
} a;
```

Ou seja, não é mais necessário o uso de typedef neste caso. A simples declaração de uma estrutura já permite que se use o nome sem a necessidade da palavra reservada struct , como mostrado abaixo:

```
struct a {};

void f(void)
{
   struct a a1;  // em C o tipo é usado como "struct a"
```

```
   a        a2;  // em C++ a palavra struct
                 // não é mais necessária
}
```

Declaração de uniões

Em C++ as uniões podem ser anônimas. Uma união anônima é uma declaração da forma:

```
union { lista dos campos };
```

Neste caso, os campos da união são usados como se fossem declarados no escopo da própria união:

```
void f(void)
{
   union { int a; char *str; };
   a = 1;
   // ...
   str = malloc( 10 * sizeof(char) );
}
```

a e *str* são campos de uma união anônima, e são usados como se tivessem sido declarados como variáveis da função. Mas na realidade as duas têm o mesmo endereço. Um uso mais comum para uniões anônimas é dentro de estruturas:

```
struct A {
   int tipo;
   union {
      int   inteiro;
      float real;
      void *ponteiro;
   };
};
```

Os três campos da união podem ser acessados diretamente, da mesma maneira que o campo *tipo*.

Protótipos de funções

Em C++ uma função só pode ser usada se esta já foi declarada. Em C, o uso de uma função não declarada geralmente causava uma warning do compilador, mas não um erro. Em C++ isto é um erro.

Para usar uma função que não tenha sido definida antes da chamada — tipicamente chamada de funções entre módulos —, é necessário usar protótipos. Os protótipos de C++ incluem não só o tipo de retorno da função, mas também os tipos dos parâmetros:

```
void f(int a, float b); // protótipo da função f

void main(void)
{
   f(1, 4.5);  // o protótipo possibilita a utilização de f aqui
}

void f(int a, float b)
{
   printf("%.2f\n", b+a+0.5);
}
```

Uma tentativa de utilizar uma função não declarada gera um erro de símbolo desconhecido.

Funções que não recebem parâmetros

Em C puro, um protótipo pode especificar apenas o tipo de retorno de uma função, sem dizer nada sobre seus parâmetros. Por exemplo,

```
float f(); // em C, não diz nada sobre os parâmetros de f
```

é um protótipo incompleto da função *f*. De acordo com a seção anterior, um protótipo incompleto não faz muito sentido. Na realidade, esta é uma das diferenças entre C e C++. Um compilador de C++ interpretará a linha acima como o protótipo de uma função que retorna um float e não recebe nenhum parâmetro. Ou seja, é exatamente equivalente a uma função (void):

```
float f(); // em C++ é o mesmo que float f(void);
```

Funções inline

Funções inline são comuns em C++, tanto em funções globais como em métodos. Estas funções têm como objetivo tornar mais eficiente, em relação à velocidade, o código que chama estas funções. Elas são

tratadas pelo compilador quase como uma macro: a chamada da função é substituída pelo corpo da função. Para funções pequenas, isto é extremamente eficiente, já que evita geração de código para a chamada e o retorno da função.

O corpo de funções inline pode ser tão complexo quanto uma função normal, não há limitação alguma. Na realidade o fato de uma função ser inline ou não é apenas uma otimização; a semântica de uma função e sua chamada é a mesma seja ela inline ou não.

Este tipo de otimização normalmente só é utilizado em funções pequenas, pois funções inline grandes podem aumentar muito o tamanho do código gerado. Do ponto de vista de quem chama a função, não faz nenhuma diferença se a função é inline ou não.

Nem todas as funções declaradas inline são expandidas como tal. Se o compilador julgar que a função é muito grande ou complexa, ela é tratada como uma função normal. No caso de o programador especificar uma função como inline e o compilador decidir que ela será uma função normal, será sinalizada uma warning. O critério de quando expandir ou não funções inline não é definido pela linguagem C++, e pode variar de compilador para compilador.

Para a declaração de funções inline foi criada mais uma palavra reservada, inline. Esta deve preceder a declaração de uma função inline:

```
inline double quadrado(double x)
{
   return x * x;
}
```

Neste caso a função *quadrado* foi declarada como inline. A chamada desta função é feita normalmente:

```
{
   double c = quadrado( 7 );
   double d = quadrado( c );
}
```

Assim como a chamada é feita da mesma maneira, o significado também é o mesmo com ou sem inline. A diferença fica por conta do código gerado. O trecho de código acima será compilado como se fosse:

```
{
   double c = 7 * 7;
   double d = c * c;
}
```

Além de ser mais eficiente, possibilita, por parte do compilador, otimizações extras. Por exemplo, calcular automaticamente, durante a compilação, o resultado de 7*7, gerando código para uma simples atribuição de 49 a *c*.

Fazendo uma comparação com as macros, que são usadas em C para produzir este efeito, nota-se que funções inline as substituem com vantagens. A primeira é que uma macro é uma simples substituição de texto, enquanto que funções inline são elementos da linguagem, e são verificadas quanto a erros. Além disso, macros não podem ser usadas exatamente como funções, como mostra o exemplo a seguir:

```
#define quadrado(x) ((x)*(x))
void main(void)
{
   double a = 4;
   double b = quadrado(a++);
}
```

Antes da compilação, a macro será expandida para:

```
double b = ((a++)*(a++));
```

A execução desta linha resultará na atribuição de 4*5 = 20 a *b*, além de incrementar duas vezes a variável *a*. Com certeza não era este o efeito desejado.

Assim como funções, métodos também podem ser declarados como inline. No caso de métodos não é necessário usar a palavra reservada inline. A regra é a seguinte: funções com o corpo declarado dentro da própria classe são tratadas como inline; funções declaradas na classe mas definidas fora não são inline.

Supondo uma classe *Pilha*, o método *vazia* é simples o suficiente para justificar uma função inline:

```
struct Pilha {
   elemPilha* topo;
   int  vazia() { return topo==NULL; }
   void push(int v);
   int  pop();
};
```

Alguns detalhes devem ser levados em consideração durante a utilização de funções inline. O que o compilador faz quando se depara com uma chamada de função inline? O corpo da função deve ser

expandido no lugar da chamada. Isto significa que o corpo da função deve estar disponível para o compilador antes da chamada.

Um módulo C++ é composto por dois arquivos, o arquivo com as declarações exportadas (.h) e outro com as implementações (.c, .cc ou .cpp). Tipicamente, uma função exportada por um módulo tem o seu protótipo no .h e sua implementação no .c. Isso porque um módulo não precisa conhecer a implementação de uma função para usá-la. Mas isso só é verdade para funções não inline. Por causa disso, funções inline exportadas precisam ser implementadas no .h e não mais no .c.

Referências

Referência é um novo modificador que permite a criação de novos tipos derivados. Assim como pode-se criar um tipo ponteiro para um inteiro, pode-se criar uma referência para um inteiro. A declaração de uma referência é análoga à de um ponteiro, usando o caracter & no lugar de *.

Uma referência para um objeto qualquer é, internamente, um ponteiro para o objeto. Mas, diferentemente de ponteiros, uma variável que é uma referência é utilizada como se fosse o próprio objeto. Os exemplos deixarão estas idéias mais claras. Vamos analisar referências em três utilizações: como variáveis locais, como tipos de parâmetros e como tipo de retorno de funções.

Uma variável local que seja uma referência deve ser sempre inicializada; a não inicialização causa um erro de compilação. Como as referências se referenciam a objetos, a inicialização não pode ser feita com valores constantes:

```
{
   int  a;       // ok, variável normal
   int& b = a;   // ok, b é uma referencia para a
   int& c;       // erro! não foi inicializada
   int& d = 12;  // erro! inicialização inválida
}
```

A variável *b* é utilizada como se fosse realmente um inteiro, não há diferença pelo fato de ela ser uma referência. Só que *b* não é um novo inteiro, e sim uma referência para o inteiro guardado em *a*. Qualquer alteração em *a* se reflete em *b* e vice versa. É como se *b* fosse um novo nome para a mesma variável *a*:

```
{
   int  a = 10;
   int& b = a;
   printf("a=%d, b=%d\n", a, b); // produz a=10, b=10
   a = 3;
   printf("a=%d, b=%d\n", a, b); // produz a=3, b=3
   b = 7;
   printf("a=%d, b=%d\n", a, b); // produz a=7, b=7
}
```

No caso de a referência ser um tipo de algum argumento de função, o parâmetro será passado por referência, algo que não existia em C e era simulado passando-se ponteiros:

```
void f(int a1, int &a2, int *a3)
{
   a1 = 1;  // altera cópia local
   a2 = 2;  // altera a variável passada (b2 de main)
  *a3 = 3;  // altera o conteúdo do endereço de b3
}

void main()
{
   int b1 = 10, b2 = 20, b3 = 30;
   f(b1, b2, &b3);
   printf("b1=%d, b2=%d, b3=%d\n", b1, b2, b3);
   // imprime b1=10, b2=2, b3=3
}
```

O efeito é o mesmo para *b2* e *b3*, mas repare que no caso de *b3* é passado o endereço explicitamente, e a função tem que tratar o parâmetro como tal.

Falta ainda analisar um outro uso de referências, quando esta aparece como tipo de retorno de uma função. Por exemplo:

```
int& f()
{
   static int global;
   return global;     // retorna uma referência para a variável
}
```

```
void main()
{
   f() = 12;         // altera a variável global
}
```

É importante notar que este exemplo é válido porque *global* é uma variável static de *f*, ou seja, é uma variável global com escopo limitado a *f*. Se *global* fosse uma variável local comum, o valor de retorno seria inválido, pois quando a função *f* terminasse a variável *global* não existiria mais, e portanto a referência seria inválida. Como um ponteiro perdido. Outras seções apresentam exemplos práticos desta utilização.

Alocação de memória

A alocação dinâmica de memória, que em C era tratada com as funções malloc e free, é diferente em C++. Programas C++ não precisam mais usar estas funções. Para o gerenciamento da memória, foram criados dois novos operadores, new e delete, que são duas palavras reservadas. O operador new aloca memória e é análogo ao malloc; delete desaloca memória e é análogo ao free. A motivação desta modificação ficará clara no estudo de classes, mais especificamente na parte de construtores e destrutores.

Por ser um operador da linguagem, não é mais necessário, como no malloc, calcular o tamanho da área a ser alocada. Outra preocupação não mais necessária é a conversão do ponteiro resultado para o tipo correto. O operador new faz isso automaticamente:

```
int * i1 = (int*)malloc(sizeof(int));      // C
int * i2 = new int;                        // C++
```

A alocação de um vetor também é bem simples:

```
int * i3 = (int*)malloc(sizeof(int) * 10); // C
int * i4 = new int[10];                    // C++
```

A liberação da memória alocada é feita pelo operador delete. Este operador pode ser usado em duas formas; uma para desalocar um objeto e outra para desalocar um vetor de objetos:

```
free(i1);          // alocado com malloc (C  )
delete i2;         // alocado com new    (C++)
free(i3);          // alocado com malloc (C  )
delete [] i4;      // alocado com new[]  (C++)
```

A utilização de delete para desalocar um vetor, assim como a utilização de delete[] para desalocar um único objeto tem consequências indefinidas. A necessidade de diferenciação se explica pela existência de destrutores, apresentados mais à frente.

Valores default para parâmetros de funções

Em C++ existe a possibilidade de definir valores default para parâmetros de uma função. Por exemplo, uma função que imprime uma string na tela em alguma posição especificada. Se nenhuma posição for especificada, a string deve ser impressa na posição do cursor:

```
void impr( char* str, int x = -1, int y = -1)
{
   if (x == -1) x = wherex();
   if (y == -1) y = wherey();
   gotoxy( x, y );
   cputs( str );
}
```

Esta definição indica que a função *impr* tem três parâmetros, sendo que os dois últimos tem valores default. Quando uma função usa valores default, os parâmetros com default devem ser os últimos. A função acima pode ser utilizada das seguintes maneiras:

```
impr( “especificando a posição”, 10, 10 ); // x=10, y=10
impr( “só x”, 20 );                        // x=20, y=-1
impr( “nem x nem y” );                     // x=-1, y=-1
```

O uso deste recurso também envolve alguns detalhes. A declaração do valor default só pode aparecer uma vez. Isto deve ser levado em consideração ao definir funções com protótipos. A maneira correta especifica o valor default apenas no protótipo:

```
void impr( char* str, int x = -1, int y = -1 );
...
void impr( char* str, int x, int y )
{
   ...
}
```

Mas a regra diz apenas que os valores só aparecem em um lugar. Nada impede a seguinte construção:

```
void impr( char* str, int x, int y );
...
 void impr( char* str, int x = -1, int y = -1 )
{
   ...
}
```

Nesse caso, antes da definição do corpo de *impr* ela será tratada como uma função sem valores default.

Sobrecarga de nomes de funções

Este novo recurso permite que um nome de função possa ter mais de um significado, dependendo do contexto. Ter mais de um significado quer dizer que um nome de função pode estar associado a várias implementações diferentes. Apesar disso soar estranho para pessoas familiarizadas com C, no dia a dia aparecem situações idênticas no uso da nossa linguagem. Consideremos o verbo tocar; podemos usá-lo em diversas situações. Podemos tocar violão, tocar um disco etc. Em C++, diz-se que o verbo tocar está sobrecarregado. Cada contexto está associado a um significado diferente, mas todos estão conceitualmente relacionados.

Um exemplo simples é uma função que imprime um argumento na tela. Seria interessante se pudéssemos usar esta função para vários tipos de argumentos, mas cada tipo exige uma implementação diferente. Com sobrecarga isto é possível; o compilador escolhe que implementação usar dependendo do contexto. No caso de funções o contexto é o tipo dos parâmetros (só os parâmetros, não os resultados):

```
display( “string” );
display( 123 );
display( 3.14159 );
```

Na realidade, a maioria das linguagens usa sobrecarga internamente, mas não deixa este recurso disponível para o programador. C é uma destas linguagens. Por exemplo:

```
a = a + 125;
b = 3.14159 + 1.17;
```

O operador + está sobrecarregado. Na primeira linha, ele se refere a uma soma de inteiros; na segunda a uma soma de reais. Estas somas exigem implementações diferentes, e é como se existissem duas funções, a primeira sendo

```
int +(int, int);
```

e a outra como

```
float +(float, float);
```

Voltando à função *display*, é preciso definir as várias implementações requeridas:

```
void display( char *v ) { printf(“%s”, v); }
void display( int   v ) { printf(“%d”, v); }
void display( float v ) { printf(“%f”, v); }
```

A simples declaração destas funções já tem todas as informações suficientes para o compilador fazer a escolha correta. Isto significa que as funções não são mais distinguidas apenas pelo seu nome, mas pelo nome e pelo tipo dos parâmetros.

A tentativa de declarar funções com mesmo nome que não possam ser diferenciadas pelo tipo dos parâmetros causa um erro:

```
void f(int a);
int  f(int b); // erro! redeclaração de f!!!
```

O uso misturado de sobrecarga e valores default para parâmetros também pode causar erros:

```
void f();
void f(int a = 0);

void main()
{
   f( 12 ); // ok, chamando f(int)
   f();     // erro!! chamada ambígua: f() ou f(int = 0)???
}
```

Repare que nesse caso o erro ocorre no uso da função, e não na declaração.

Parâmetros de funções não utilizados

A maioria dos compiladores gera uma warning quando um dos parâmetros de uma função não é utilizado no seu corpo. Por exemplo:

```
void f(int a)
{
```

```
   return 1;
}
```

Este é um caso típico. O aviso do compilador faz sentido, já que a não utilização do parâmetro a pode ser um erro na lógica da função. Mas e se for exatamente esta a implementação da função *f*? À primeira vista pode parecer estranho uma função que não utilize algum de seus parâmetros, mas em programação com callbacks (isto ficará mais claro a partir da introdução de programação orientada a eventos) é comum as funções ignorarem alguns parâmetros. Se for este o caso, não é necessário conviver com as warnings do compilador. Basta omitir o nome do parâmetro que não será usado:

```
void f(int)
{
   return 1;
}
```

Esta é uma função que recebe um inteiro como parâmetro, mas este inteiro não é utilizado na implementação da função.

Operador de escopo

C++ possui um novo operador que permite o acesso a nomes declarados em escopos que não sejam o corrente. Por exemplo, considerando o seguinte programa C:

```
char *a;

void main(void)
{
   int a;
   a = 23;
   /* como acessar a variável global a??? */
}
```

A declaração da variável local *a* esconde a global. Apesar de a variável *a* global existir, não há como referenciar o seu nome, pois no escopo da função *main*, o nome *a* está associado à local *a*.

O operador de escopo possibilita o uso de nomes que não estão no escopo corrente, o que pode ser usado neste caso:

```
char *a;

void main(void)
{
   int a;
   a = 23;
   ::a = “abc”;
}
```

A sintaxe deste operador é a seguinte:

```
escopo::nome,
```

onde *escopo* é o nome da classe onde está declarado no nome *nome*. Se *escopo* não for especificado, como no exemplo acima, o nome é procurado no espaço global.

Outros usos deste operador são apresentados nas seções sobre classes aninhadas, campos de estruturas static e métodos static.

Incompatibilidades entre C e C++

Teoricamente, um programa escrito em C é compatível com um compilador C++. Na realidade esta compatibilidade não é de 100%, pois só o fato de C++ ter mais palavras reservadas já inviabiliza esta compatibilidade total. Outros detalhes contribuem para que um programa C nem sempre seja também um programa C++ válido.

Serão discutidos os pontos principais destas incompatibilidades; tanto as diferenças na linguagem como a utilização de códigos C e C++ juntos em um só projeto. No entanto a discussão não será extensiva, pois algumas construções suspeitas podem levar a várias pequenas incompatibilidades, como por exemplo:

```
a = b //* comentário
          etc.       */ 2;
```

Extraídos os comentários, este código em C fica sendo:

```
a = b / 2;
```

Enquanto que em C++, o resultado é diferente:

```
a = b
etc. */ 2;
```

Casos como estes são incomuns e não vale a pena analisá-los.

Palavras reservadas

Talvez a maior incompatibilidade seja causada pelo simples fato que C++ tem várias outras palavras reservadas. Isto significa que qualquer programa C que declare um identificador usando uma destas palavras não é um programa C++ correto. Aqui está uma lista com as novas palavras reservadas[†]:

```
catch      new        template
class      operator   this
delete     private    throw
friend     protected  try
inline     public     virtual
```

A única solução para traduzir programas que fazem uso destas palavras é trocar os nomes dos identificadores, o que pode ser feito usando diretivas #define.

Exigência de protótipos

Um dos problemas que pode aparecer durante a compilação de um programa C está relacionado ao fato de que protótipos não são obrigatórios em C. Como C++ exige protótipos, o que era uma warning C pode se transformar em um erro C++.

Estes erros simplesmente forçam o programador a fazer uma coisa que já devia ter sido feita mesmo com o compilador C: usar protótipos para as funções.

```
void main(void)
{
   printf("teste"); // C:   warning! printf undeclared
                    // C++: error!   printf undeclared
}
```

Funções que não recebem parâmetros

Programas que utilizam protótipos incompletos, ou seja, protótipos que só declaram o tipo de retorno de uma função, podem causar erros em C++. Este tipo de protótipo é considerado obsoleto em C, mas ainda válido. Em C++, estes protótipos são interpretados como declarações de funções que não recebem parâmetros, o que certamente causará erros se a função receber algum:

```
float square();  // C:   protótipo incompleto, não se sabe nada
                 //      sobre os parâmetros
                 // C++: protótipo completo,
                 //      não recebe parâmetros

void main(void)
{
   float a = square(3); // C:   ok
                        // C++: error! too many arguments
}
```

Estruturas aninhadas

Outra incompatibilidade pode aparecer em programas C que declaram estruturas aninhadas. Em C, só há um escopo para os tipos: o global. Por esse motivo, mesmo que uma estrutura tenha sido declarada dentro de outra, ou seja, aninhada, ela pode ser usada como se fosse global. Por exemplo:

```
struct  S1 {
   struct S2 {
      int a;
   } b;
};

void main(void)
{
   struct S1 var1;
```

[†] Esta lista não é definitiva, já que constantemente novos recursos vão sendo adicionados ao padrão.

```
    struct S2 var2;  // ok em C, errado em C++
}
```

Este é um programa correto em C. Em C++, uma estrutura só é válida no escopo em que foi declarada. No exemplo acima, por ter sido declarada dentro de *S1*, a estrutura *S2* só pode ser usada dentro de *S1*. Nesse caso, a tentativa de declarar a variável *var2* causaria um erro, pois *S2* não é um nome global.

Em C++, *S2* pode ser referenciada utilizando-se o operador de escopo:

```
struct S1::S2 var2; // solução em C++
```

Uso de bibliotecas C em programas C++

Esta seção discute um problema comum ao se tentar utilizar bibliotecas C em programas C++. A origem do problema é o fato de C++ permitir sobrecarga de nomes de funções. Vamos analisar o trecho de código abaixo:

```
void f(int);

void main(void)
{
   f(12);
}
```

O programa acima declara uma função *f* que será chamada com o argumento 12. O código final gerado para a chamada da função terá uma linha assim:

```
call nome-da-função-escolhida
```

Este nome deve bater com o nome gerado durante a compilação da função *f*, seja qual for o módulo a que ela pertence. Caso isto não aconteça, a linkedição do programa sinalizará erros de símbolos indefinidos. Isto significa que existem regras para a geração dos nomes dos identificadores no código gerado. Em C, esta regra é simples: os símbolos têm o mesmo nome do código fonte com o prefixo _. No caso acima, o código gerado seria:

```
call _f
```

Entretanto, C++ não pode usar esta mesma regra simples, porque um mesmo nome pode se referir a várias funções diferentes pelo mecanismo de sobrecarga. Basta imaginar que poderíamos ter:

```
void f(int);
void f(char*);
```

Nesse caso a regra de C não é suficiente para diferenciar as duas funções. O resultado é que C++ tem uma regra complexa para geração dos nomes, que envolve codificação dos tipos também. O compilador Borland C++ 4.0 codifica as funções acima, respectivamente, como:

```
@f$qi
@f$qpc
```

Isto deve ser levado em consideração na compilação da chamada da função. A chamada de *f* com o argumento 12 pode gerar dois códigos diferentes. Se a função *f* for na realidade uma função compilada em C, o seu nome será *_f*, enquanto que se esta for uma função C++, o seu nome será *@f$qi*. Qual deve ser então o código gerado? Qual nome deve ser chamado?

A única maneira de resolver este problema é indicar, no protótipo da função, se esta é uma função C ou C++. No exemplo acima, o compilador assume que a função é C++. Se *f* for uma função C, o seu protótipo deve ser:

```
extern "C" void f(int);
```

A construção extern "C" pode ser usada de outra maneira, para se aplicar a várias funções de uma vez:

```
extern "C"{
   void f(int);
   void f(char);
   // ...
}
```

Dentro de um bloco extern pode aparecer qualquer tipo de declaração, não apenas funções. A solução então é alterar os header files das bibliotecas C, explicitando que as funções não são C++. Para isso, basta envolver todas as declarações em um bloco destes.

Como uma declaração extern "C" só faz sentido em C++, ela causa um erro de sintaxe em C. Para que os mesmos header files sejam usados em programas C e C++, basta usar a macro pré-definida __cplusplus, definida em todo compilador C++. Com estas alterações, um header file de uma biblioteca C terá a seguinte estrutura:

```
#ifdef __cplusplus
extern "C" {
#endif
```

```
// todas as declarações
// ...

#ifdef __cplusplus
}
#endif
```

Exercício 2 - Calculadora RPN com novos recursos de C++

Analisar a calculadora alterando-a de acordo com os novos recursos de C++

Recursos de C++ relacionados às classes

Classes aninhadas

Declarações de classes podem ser aninhadas. Uma classe aninhada não reside no espaço de nomes global, como era em C. Para referenciar uma classe aninhada fora do seu escopo, é preciso usar o operador de escopo.

Este aninhamento é usado para encapsular classes auxiliares que só fazem sentido dentro de um escopo reduzido.

```
struct A {
   struct B {
      int b;
   };
   int a;
};

void main()
{
   A a;
   A::B b;
}
```

O código acima simplesmente declara a estrutura B dentro de A; não existe nenhum campo de A que seja do tipo B.

Declaração incompleta

Em C, a declaração de estruturas mutuamente dependentes é feita naturalmente:

```
struct A { struct B *next; };
struct B { struct A *next; };
```

Em C++ a construção acima pode ser usada. No entanto, como a declaração de uma estrutura em C++ já define o nome como um tipo sem necessidade de typedefs, é normal em C++ o uso de tipos sem o uso da palavra reservada struct:

```
struct A { B *next; }; // Erro! B indefinido
struct B { A *next; };
```

No caso de tipos mutuamente dependentes, isto só é possível usando uma declaração incompleta:

```
struct B;  // declaração incompleta de B
struct A { B *next; };
struct B { A *next; };
```

Assim como em C, antes da declaração completa da estrutura só é possível usar o nome para declarar ponteiros e referências deste tipo. Como as informações estão incompletas, não é possível declarar variáveis deste tipo nem utilizar campos da estrutura.

Métodos const

Métodos const são métodos que não alteram o estado interno de um objeto. Assim como era possível declarar variáveis e parâmetros const em C, é possível declarar um método const em C++. A especificação const faz parte da declaração do método.

A motivação para esta nova declaração pode ser esclarecida com o exemplo abaixo:

```
struct A {
   int value;
   int get()        { return value; }
   void put (int v) { value = v; }
```

```
};

void f(const A a)
{
   // parâmetro a não pode ser modificado
   // quais métodos de a podem ser chamados???
   int b = a.get(); // Erro! método não é const
}
```

A função *f* não pode alterar o seu parâmetro por causa da declaração const. Isto significa que os campos de *a* podem ser lidos mas não podem ser modificados. E quanto aos métodos? O que define quais métodos podem ser chamados é a sua implementação. Esta verificação não pode ser feita pelo compilador, já que na maioria das vezes o código dos métodos não está disponível. É responsabilidade do programador declarar quais métodos não modificam o estado interno do objeto. Estes métodos podem ser aplicados a objetos declarados const.

No exemplo acima, o método *get* pode ser declarado const, o que possibilita a sua chamada na função *f*. Se o método *put* for declarado const, o compilador acusa um erro de compilação pois o campo *value* é modificado em sua implementação:

```
struct A {
   int value;
   int get() const  { return value; }
   void put (int v) { value = v; }
};

void f(const A a)
{
   // parâmetro a não pode ser modificado
   int b = a.get(); // ok, método const
}
```

this

Em todo método não static (ver seção sobre métodos static), a palavra reservada this é um ponteiro para o objeto sobre o qual o método está executando.

Todos os métodos de uma classe são sempre chamados associados a um objeto. Durante a execução de um método, os campos do objeto são manipulados normalmente, sem necessidade de referência ao objeto. E se um método precisar acessar não os campos de um objeto, mas o próprio objeto? O exemplo abaixo ilustra este caso:

```
struct A {
   int i;
   A& inc();
};

// Este método incrementa o valor interno
// e retorna o próprio objeto
A& A::inc()
{
   // estamos executando este código para obj1 ou obj2?
   // como retornar o próprio objeto?
   i++;
   return *this;  // this aponta para o próprio objeto
}

void main()
{
   A obj1, obj2;
   obj1.i = 0;
   obj2.i = 100;
   for (j=0; j<10; j++)
      printf("%d\n", (obj1.inc()).i);
}
```

O tipo de this dentro de um método de uma classe *X* é

```
X* const
```

a não ser que o método seja declarado const. Nesse caso, o tipo de this é:

```
const X* const
```

Campos de estrutura static

Em C++, membros de uma classe podem ser static. Quando uma variável é declarada static dentro de uma classe, todas as instâncias de objetos desta classe compartilham a mesma variável. Uma variável static é uma variável global, só que com escopo limitado à classe.

A declaração de um campo static aparece na declaração da classe, junto com todos os outros campos. Como a declaração de uma classe é normalmente incluída em vários módulos de uma mesma aplicação via *header files* (arquivos .h), a declaração dentro da classe é equivalente a uma declaração de uma variável global extern. Ou seja, a declaração apenas diz que a variável existe; algum módulo precisa defini-la. O exemplo abaixo ilustra a utilização de campos static:

```
struct A {
   int a;
   static int b; // declara a variável,
                 // equivalente ao uso de extern para
                 // variáveis globais
};

int A::b = 0;    // define a variável criando o seu espaço
                 // esta é a hora de inicializar

void main(void)
{
   A a1, a2;
   a1.a = 0;  // modifica o campo a de a1
   a1.b = 1;  // modifica o campo b compartilhado por a1 e a2
   a2.a = 2;  // modifica o campo a de a1
   a2.b = 3;  // modifica o campo b compartilhado por a1 e a2
   printf("%d %d %d %d", a1.a, a1.b, a2.a, a2.b);
   // imprime 0 3 2 3
}
```

Se a definição

```
int A::b = 0;
```

for omitida, o arquivo é compilado mas o linker acusa um erro de símbolo não definido.

Como uma variável estática é única para todos os objetos da classe, não é necessário um objeto para referenciar este campo. Isto pode ser feito com o operador de escopo:

```
A::b = 4;
```

Métodos static

Assim como campos static são como variáveis globais com escopo reduzido à classe, métodos static são como funções globais com escopo reduzido à classe. Isto significa que métodos static não tem o parâmetro implícito que indica o objeto sobre o qual o método está sendo executado (this), e portanto apenas os campos static podem ser acessados:

```
struct A {
   int a;
   static int b;
   static void f();
};

int A::b = 10;

void A::f()
{
   a = 10; // errado, a só faz sentido com um objeto
   b = 10; // ok, b declarado static
}
```

Ponteiros para métodos

É possível obter o ponteiro para um método. Isto pode ser feito aplicando o operador & a um nome de método totalmente qualificado, ou seja:

```
& classe::método
```

Uma variável do tipo ponteiro para um membro da classe *X* é obtida com o declarador *X::**:

```
struct A {
   void f(int);
};

void main()
{
   void (A::*p) (int); // declara a variável p
   p = &A::f;          // obtém o ponteiro para f

   A  a1;
   A* a2 = new A;

   (a1.*p)(10);          // aplica p a a1
   (a2->*p)(20);         // aplica p a a2
}
```

A aplicação dos métodos é feita utilizando-se os novos operadores .* e ->*.

Capítulo 3

Encapsulamento

Como foi visto anteriormente, um TAD é definido pela sua interface, ou seja, como ele é manipulado. Um TAD pode ter diversas implementações possíveis, e, independentemente desta ou daquela implementação, objetos deste tipo são usados sempre da mesma forma. Os atributos além da interface, ou seja, os atributos dependentes da implementação, não precisam e não devem estar disponíveis para o usuário de um objeto, pois este deve ser acessado exclusivamente através da interface definida. O ato de esconder estas informações é chamado de encapsulamento.

Os mecanismos apresentados até aqui permitem a definição da interface em um tipo, permitindo códigos bem mais modulares e organizados. No entanto, não permitem o encapsulamento de dados e/ou código.

Controle de acesso - public e private

Parte do objetivo de uma classe é esconder o máximo de informação possível. Então é necessário impor certas restrições na maneira como uma classe pode ser manipulada, e que dados e código podem ser usados. Podem ser estabelecidos três níveis de permissão de acordo com o contexto de uso dos atributos:

- nos métodos da classe
- a partir de objetos da classe
- métodos de classes derivadas (este ponto será visto posteriormente)

Cada um destes três contextos tem privilégios de acesso diferenciados; cada um tem uma palavra reservada associada, private, public e protected respectivamente.

O exemplo abaixo ilustra o uso destas novas palavras reservadas:

```
struct controle {
 private:
   int a;
   int f1( char* b );
 protected:
   int b;
   int f2( float );
 public:
   int c;
   float d;
   void f3( controle* );
};
```

As seções podem ser declaradas em qualquer ordem, inclusive podem aparecer mais de uma vez. O exemplo abaixo é equivalente ao apresentado acima:

```
struct controle {
   int c;
   int d;
 private:
   int a;
 protected:
   int b;
 protected:
   int f2( float );
 public:
   void f3( controle* );
 private:
   int f1( char* b );
};
```

Atributos private são os mais restritos. Somente a própria classe pode acessar os atributos privados. Ou seja, somente os métodos da própria classe tem acesso a estes atributos.

```
struct SemUso {
 private:
   int valor;
```

```
   void f1() { valor = 0; }
   int  f2() { return valor; }
};

int main()
{
   SemUso p; // cria um objeto do tipo SemUso
   p.f1();                  // erro, f1 é private!
   printf("%d", p.f2() );   // erro, f2 é private
   return 0;
}
```

No exemplo anterior, todos os membros são privados, o que impossibilita o uso de um objeto da classe *SemUso*. Para que as funções *f1* e *f2* pudessem ser usadas, elas precisariam ser públicas. O código a seguir é uma modificação da declaração da classe *SemUso* para tornar as funções *f1* e *f2* públicas e portanto passíveis de serem chamadas em *main*.

```
struct SemUso {
 private:
   int valor;
 public:
   void f1() { valor = 0; }
   int  f2() { return valor; }
};
```

Membros protected serão explicados quando forem introduzidas as classes derivadas.

Para exemplificar melhor o uso do controle de acesso, vamos considerar uma implementação de um conjunto de inteiros. As operações necessárias são, por exemplo, inserir e retirar um elemento, verificar se um elemento pertence ao conjunto e a cardinalidade do conjunto. Então o nosso conjunto terá pelo menos o seguinte:

```
struct Conjunto {
   void insere(int n);
   void retira(int n);
   int  pertence(int n);
   int  cardinalidade();
};
```

Se a implementação deste conjunto usar listas encadeadas, é preciso usar uma estrutura auxiliar *elemento* que seriam os nós da lista. A nova classe teria ainda uma variável private que seria o ponteiro para o primeiro elemento da lista. Outra variável private seria um contador de elementos. Vamos acrescentar ainda um método para limpar o conjunto, para ser usado antes das funções do conjunto propriamente ditas. Eis a definição da classe:

```
struct Conjunto {
 private:
   struct  listElem {
      listElem *prox;
      int valor;
   };
   listElem* lista;
   int nElems;
 public:
   void limpa() { nElems=0; lista=NULL; }
   void insere(int n);
   void retira(int n);
   int  pertence(int n);
   int  cardinalidade();
};
```

Como somente os métodos desta classe tem acesso aos campos privados, a estrutura interna não pode ser alterada por quem usa o conjunto, o que garante a consistência do conjunto.

Declaração de classes com a palavra reservada class

As classes podem ser declaradas usando-se a palavra reservada class no lugar de struct. A diferença entre as duas opções é o nível de proteção caso nenhum dos especificadores de acesso seja usado. Os membros de uma struct são públicos, enquanto que em uma class, os membros são privados:

```
struct A {
   int a;  // a é público
};
```

```
class B {
   int a;  // a é privado
};
```

Como as interfaces das classes devem ser as menores possíveis e devem ser também explícitas, as declarações com a palavra class são mais usadas. Com class, somente os nomes explicitamente declarados como *public* são exportados.

A partir de agora os exemplos vão ser feitos usando-se a palavra class.

Classes e funções friend

Algumas vezes duas classes são tão próximas conceitualmente que seria desejável que uma delas tivesse acesso irrestrito aos membros da outra.

No exemplo anterior, não há meio de percorrer o conjunto para, por exemplo, imprimir todos os elementos. Para fazer isto, seria preciso ter acesso ao dado *lista* e à estrutura *listElem*, ambos private. Uma maneira de resolver este problema seria aumentar a interface do conjunto oferecendo funções para percorrer os elementos. Esta solução seria artificial, pois estas operações não fazem parte do TAD conjunto.

A solução normalmente usada é a idéia de iteradores. Um iterador atua sobre alguma coleção de elementos e a cada chamada retorna um elemento diferente da coleção, até que já tenha retornado todos. Um iterador para o nosso conjunto seria:

```
class IteradorConj {
   Conjunto::listElem *corrente;
 public:
   void inicia( Conjunto* c ) { corrente = c->lista; }
   int terminou()             { return corrent==NULL; }
   int proxElem()
   { int n=corrente->valor; corrent=corrente->prox; return n; }
};

void main()
{
   Conjunto conj;  // cria conjunto
   conj.limpa();  // inicializa para operações
   conj.insere(10); //insere o elemento 10
   // ...
   IteradorConj it; // cria um iterador
   it.inicia( &conj ); // inicializa o iterador com conj
   while (!it.terminou())   // percorre todos os elementos
      printf("%d\n", it.proxElem() );  // imprimindo-os
}
```

O problema aqui é que o iterador usa dados privados de *Conjunto*, o que gera erros durante a compilação. Esse é um caso em que as duas classes estão intimamente ligadas, e então seria conveniente, na declaração de *Conjunto*, dar acesso irrestrito à classe *IteradorConj*.

Para isso existe a palavra reservada friend. Ela serve para oferecer acesso especial à algumas classes ou funções. Na declaração da classe Conjunto é possível dar permissão irrestrita aos seus atributos. A classe *Conjunto* chega à sua forma final:

```
class Conjunto {
   friend class IteradorConj;

   struct  listElem {
   ...
   ...
```

Também é possível declarar funções friend. Nesse caso, a função terá acesso irrestrito aos componentes da classe que a declarou como friend. Exemplo de função friend:

```
class No {
   friend int leValor( No* ); // dá acesso privilegiado
                              // à função leValor
   int valor;
 public:
   void setaValor( int v ) { valor=v; }
};
```

```
int leValor( No* n )
{
   return n->valor; // acessa dado private de No
}
```

O recurso de classes e funções friend devem ser usados com cuidado, pois isto é um furo no encapsulamento dos dados de uma classe. Projetos bem elaborados raramente precisam lançar mão de classes ou funções friend.

Exercício 3 - Calculadora RPN com controle de acesso

Modificar a calculadora de acordo com o controle de acesso

Construtores e destrutores

Como o nome já indica, um construtor é uma função usada para construir um objeto de uma dada classe. Ele é chamado automaticamente assim que um objeto é criado. Analogamente, os destrutores são chamados assim que os objetos são destruídos.

Assim como o controle de acesso desempenha um papel importante para manter a consistência interna dos objetos, os construtores são fundamentais para garantir que um objeto recém criado esteja também consistente.

No exemplo onde foi implementada a classe *Conjunto*, foi necessária a introdução de uma função, *limpa*, que precisava ser chamada para inicializar o estado do objeto. Se esta função não for chamada antes da manipulação de cada objeto, os resultados são imprevisíveis (os campos *nElems* e *lista* contém lixo). Deixar esta chamada a cargo do programador é aumentar o potencial de erro do programa. Na realidade, a função *limpa* deveria ser um construtor de *Conjunto*. O compilador garante que o construtor é a primeira função a ser executada sobre um objeto. A mesma coisa acontecia em *IteradorConj*. A função *inicia* deveria ser um construtor, pois antes desta chamada o estado do objeto é imprevisível.

Destrutores são normalmente utilizados para liberar recursos alocados pelo objeto, como memória, arquivos etc.

Declaração de construtores e destrutores

Os construtores e destrutores são métodos especiais. Nenhum dos dois tem um tipo de retorno, e o destrutor não pode receber parâmetros, ao contrário do construtor.

A declaração de um construtor é feita definindo-se um método com o mesmo nome da classe. O nome do destrutor é o nome da classe precedido de ~ (til). Ou seja, um construtor de uma classe *X* é declarado como um método de nome *X*, o nome do destrutor é *~X*; ambos sem tipo de retorno. O destrutor não pode ter parâmetros; o construtor pode. Pelo mecanismo de sobrecarga (funções são diferenciadas não apenas pelo nome, mas pelos parâmetros também), classes podem ter mais de um construtor.

Com o uso de construtores, as classe *Conjunto* e *IteradorConj* passa a ser:

```
class Conjunto {
   friend class IteradorConj;
   struct  listElem {
      listElem *prox;
      int valor;
   };
   listElem* lista;
   int nElems;
 public:
   Conjunto()  { nElems=0; lista=NULL; }
   ~Conjunto();  // desaloca os nós da lista
   void insere(int n);
   void retira(int n);
   int  pertence(int n);
   int  cardinalidade();
};

class IteradorConj {
   Conjunto::listElem *corrente;
 public:
   IteradorConj( Conjunto* c ) { corrente = c->lista; }
   int terminou()              { return corrent==NULL; }
   int proxElem()
```

```
   { int n=corrente->valor; corrent=corrente->prox; return n; }
};
```

Chamada de construtores e destrutores

Os construtores são chamados automaticamente sempre que um objeto da classe for criado. Ou seja, quando a variável é declarada (objetos alocados na pilha) ou quando o objeto é alocado com new (objetos dinâmicos alocados no heap).

Os destrutores de objetos alocados na pilha são chamados quando o objeto sai do seu escopo. O destrutor de objetos alocados com new só é chamado quando estes são desalocados com delete:

```
struct A {
   A()  { printf("construtor\n"); }
   ~A() { printf("destrutor\n");  }
};

void main()
{
   A a1;             // chama construtor de a1
   {
      A  a2;         // chama construtor de a2
      A *a3 = new A; // chama construtor de a3
   }                 // chama destrutor  de a2
   delete a3;        // chama destrutor  de a3
}                    // chama destrutor  de a1
```

É interessante observar que os objetos globais já estão inicializados quando a função *main* começa a ser executada. Isto significa que os construtores de objetos globais são chamados antes de *main*:

```
A global;

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

Este código produz a seguinte saída:

```
construtor
main
destrutor
```

Construtores com parâmetros

No exemplo sobre conjuntos, a classe *IteradorConj* possui um construtor que recebe um parâmetro. Se os construtores existem para garantir a consistência inicial dos objetos, o código está indicando que, para que um iterador esteja consistente, é necessário fornecer um conjunto sobre o qual será feita a iteração.

Se for possível criar um *IteradorConj* sem fornecer este conjunto, então o construtor não está garantindo nada. Mas não é isto que acontece. A declaração de um construtor impõe que os objetos só sejam criados através deles. Sendo assim, não é mais possível criar um *IteradorConj* sem fornecer um *Conjunto*. O exemplo abaixo utiliza estas classes e mostra como os construtores são chamados:

```
void main()
{
   Conjunto conj;
   conj.insere(10);
   // ...
   IteradorConj i; // erro! é obrigatório fornecer o parâmetro
   IteradorConj it( &conj ); // cria um iterador passando conj
   while (!it.terminou())             // percorre os elementos
      printf("%d ", it.proxElem() );  // imprimindo-os
}
```

Caso o iterador seja alocado dinamicamente (com new), a sintaxe é a seguinte:

```
IteradorConj *i = new IteradorConj(&conj)
```

Não é possível alocar um vetor de objetos passando parâmetros para o construtor. Por exemplo, não é possível criar um vetor de iteradores:

```
Conjunto c;
IteradorConj it(&c)[10]; // erro!!!
IteradorConj *pit;
pit = new(&c)[20];       // erro!!!
```

Construtores gerados automaticamente

O fato de algumas classes não declararem construtores não significa que elas não tenham construtores. Na realidade, o compilador gera alguns construtores automaticamente.

Um dos construtores só é gerado se a classe não declarar nenhum. Este é o construtor vazio, que permite que os objetos sejam criados. Por exemplo, como a classe

```
class X {
   int a;
};
```

não declara nenhum construtor, o compilador automaticamente gera um construtor vazio público para esta classe. A declaração abaixo é equivalente:

```
class X {
   int a;
   public: X() {} // construtor vazio
};
```

Outro construtor gerado é o construtor de cópia. Este é gerado mesmo que a classe declare algum outro construtor. O construtor de cópia de uma classe recebe como parâmetro uma referência para um objeto da própria classe. A classe *X* acima possui este construtor:

```
class X {
   int a;
   // public: X() {}  construtor vazio
   // public: X(const X&);  construtor de cópia
};
```

O construtor de cópia constrói um objeto a partir de outro do mesmo tipo. O novo objeto é uma cópia byte a byte do objeto passado como parâmetro. Repare que a existência deste construtor não é um furo na consistência dos objetos, já que ele só pode ser usado a partir de um objeto existente, e portanto, consistente. Este construtor pode ser chamado de duas formas:

```
void main()
{
   X a1;      // usa construtor vazio
   X a2(a1);  // usa construtor de cópia
   X a3 = a2; // usa construtor de cópia
}
```

A atribuição só chama o construtor de cópia quando usada junto com a declaração do objeto. É importante notar que este construtor pode ser redefinido, basta declarar um construtor com a mesma assinatura (ou seja, recebendo como parâmetro uma referência para um objeto da própria classe).

Objetos temporários

Assim como não é preciso declarar uma variável dos tipos primitivos sempre que se quer usar um valor temporariamente, é possível criar objetos temporários em C++. Uma utilização típica é a seguinte: quando uma função aceita como parâmetro um inteiro, e você quer chamá-la passando o valor 10, não é necessário atribuir 10 a uma variável apenas para chamar a função. O mesmo deve se aplicar a tipos definidos pelo usuário (classes):

```
class A {
 public:
   A(int);
   ~A();
};

void f(A);

void main()
{
   A a1(1);
   f(a1);
   f(A(10)); // cria um objeto temporário do tipo A
             // passando 10 para o construtor
             // após a chamada a f o objeto é destruído
}
```

Conversão por construtores

Um construtor com apenas um parâmetro pode ser visto como uma função de conversão do tipo do parâmetro para o tipo da classe. Por exemplo,

```
class A {
 public:
   A(int);
   A(char*, int = 0);
};

void f(A);

void main()
{
   A a1 = 1;      // a1 = A(1)
   A a2 = “abc”; // a2 = A(“abc”, 0)
   a1 = 2;        // a1 = A(2)
   f(3);          // f(A(3))
}
```

Esta conversão só é feita em um nível. Ou seja, se o construtor da classe *A* não aceita um determinado tipo, não é feita uma tentativa de converter via outros construtores o tipo dado para o tipo aceito pelo construtor:

```
class A {
  public: A(int);
};

class B {
  public: B(A);
};

B a = 1; // erro: B(A(1)) não é tentado
```

No exemplo acima, pelo menos uma conversão deve ser feita explicitamente:

```
B a = A(1)
```

Construtores privados

Os construtores, assim como qualquer método, podem ser privados. Como o construtor é chamado na criação, os objetos só poderão ser criados com este construtor dentro de métodos da própria classe ou em classes e funções friend.

Destrutores privados

Destrutores também podem ser privados. Isto significa que objetos desta classe só podem ser destruídos onde os destrutores podem ser chamados (métodos da própria classe, classes e funções friend). Usando este recurso, é possível projetar classes cujos objetos não são nunca destruídos. Outra possibilidade é o projeto de objetos que não podem ser alocados na pilha, apenas dinamicamente. Exemplo:

```
class A {
  ~A() {}
 public:
   int a;
};

void main()
{
   A a1;          // erro! destrutor privado,
                  // não pode ser chamado
                  // quando o objeto sair do escopo
   A* a2 = new A; // ok, só não pode usar delete depois
}
```

No exemplo acima, o destrutor privado impõe duas restrições: objetos não podem ser alocados na pilha e, mesmo que sejam criados dinamicamente, não podem nunca ser destruídos. Para permitir a destruição dos objetos basta criar um método que faça isso:

```
class A {
  ~A() {}
 public:
   int a;
   void destroy() { delete this; }
};
```

Inicialização de campos de classes com construtores

Quando um objeto não tem um construtor sem parâmetros, é preciso passar obrigatoriamente valores como os parâmetros. No caso do objeto ser uma variável, basta definir os parâmetros na hora da declaração:

```
class A {
   public: A(int);
};

A a(123);
```

Mas e se o objeto for um campo de uma outra classe? Nesse caso ele estará sendo criado quando um objeto desta outra classe for criado, e nessa hora os parâmetros precisarão estar disponíveis:

```
class A {
   public: A(int);
};

class B {
   A a;
};
```

Ao se criar um objeto do tipo *B*, que inteiro deve ser passado ao campo *a*? Nesse caso, o construtor de *B* tem que especificar este inteiro, e o compilador não gera um construtor vazio. Ou seja, a classe *B* tem que declarar um construtor para que seja possível criar objetos deste tipo. A sintaxe é a seguinte:

```
class B {
   A a;
 public:
   B() : a(3) {}
};
```

Exercício 4 - Calculadora RPN com construtores

Utilizar construtores na calculadora

Capítulo 4

Sobrecarga de operadores

O uso de funções sobrecarregadas não só uniformiza chamadas de funções para diferentes objetos como também permite que os nomes sejam mais intuitivos. Se um dos objetivos da sobrecarga é permitir que as funções sejam chamadas pelo nome mais natural possível, não importa se o nome já foi usado, porque não deixar o programador sobrecarregar também os operadores?

Na realidade, um operador executa algum código com alguns parâmetros, assim como qualquer função. A aplicação de um operador é equivalente à chamada de uma função. Em C++ é permitido sobrecarregar um operador, com o objetivo de simplificar a notação e uniformizar a expressão.

Existem duas maneiras de implementar operadores para classes de objetos: como funções membro e como funções globais. Por exemplo, dado um objeto w e um operador unário !, a expressão

```
!w
```

é equivalente às chamadas de funções

```
w.operator!();   // usando uma função membro
operator!(w);    // usando uma função global
```

Vejamos agora como seria com um operador binário, por exemplo, &. A expressão

```
x & y
```

é equivalente às chamadas

```
x.operator&(y); // usando uma função membro
operator&(x,y); // usando uma função global
```

Um detalhe importante é que uma função *y.operator&(x)* nunca será considerada pelo compilador para resolver a expressão *x&y*, já que isto implicaria que o operador é comutativo.

Antes do primeiro exemplo, precisamos ter em mente que C++ não permite a criação de novos operadores; só podem ser redefinidos os operadores que já existem na linguagem. Isto implica que, por exemplo, o operador / será sempre binário. Outra característica é que a prioridade também não pode ser alterada, é preservada a original do C.

Um número complexo pode ser modelado com uma classe que permita que as operações matemáticas sobre ele sejam feitas da mesma maneira que os tipos primitivos, ou seja, com os operadores +, - etc. Uma possibilidade seria:

```
class Complex {
 public:
   Complex operator+ (const Complex&);
   Complex operator- (const Complex&);
};
```

Com estes operadores, é possível fazer:

```
void main()
{
   Complex c1, c2, c3;
   c1 = c2 + c3;
}
```

Exercício 5 - Classe Complex

Implementar uma classe que represente um número complexo.

Operadores como funções globais

Suponhamos que a classe que modela complexos possui um construtor da forma:

```
Complex(float re = 0.0, float im = 0.0);
```

Este construtor permite criar um complexo especificando suas partes real e imaginária, só especificando a parte real ou ainda sem dizer nada. Em particular, este construtor define como converter um float em um complexo, o que é equivalente a chamar o construtor com apenas um parâmetro. Esta conversão permite expressões do tipo:

```
c1 = c2 + 3.0; // equivalente a c1 = c2 + Complex(3.0, 0.0)
```

Considerando que o operador é uma função, esta expressão poderia ser vista como:

```
c1 = c2.operator+(Complex(3.0,0.0));
```

A conversão foi feita porque a função operator+ espera um *Complex*, e o valor era um float. Nesse caso o compilador converte automaticamente o valor. Mas e se a expressão for a seguinte:

```
c1 = 3.0 + c2;
```

Nesse caso 3.0 não é parâmetro de função nenhuma, então a conversão não é feita. Para possibilitar esta expressão, seria preciso converter o valor explicitamente:

```
c1 = Complex(3.0) + c2;
```

No entanto, se 3.0 fosse o parâmetro para alguma função, o compilador saberia fazer a conversão. Lembrando que os operadores podem ser definidos como métodos ou como funções globais, é possível tornar 3.0 um parâmetro. É o caso de definir o operador como uma função global:

```
Complex operator+ (const Complex&, const Complex&);
```

Com esta função definida, os dois operandos passam a ser parâmetros, e ambos podem ser convertidos automaticamente.

Operadores que podem ser redefinidos

A maior parte dos operadores podem ser sobrecarregados. São eles:

```
new delete
+ - * / % ^& | ~
! = < > += -= *= /= %=
^= &= |= << >> >>= <<= == !=
<= >= && || ++ -- , ->* ->
() []
```

Tanto as formas unárias como as binárias de

```
+ - * &
```

podem ser sobrecarregadas, assim como as formas pré-fixadas ou pós fixadas de

```
++ --
```

Os seguintes operadores não podem ser sobrecarregados:

```
. .* :: sizeof ?:
```

já que estes operadores já têm um significado predefinido (exceto ?:) para objetos de qualquer classe.

A função de atribuição operator=() é definida por default para todos os objetos como a atribuição byte a byte dos campos do objeto.

Exemplo de redefinição do operador [] - classe Vector

As estratégias usadas para redefinições variam muito de acordo com o operador. Esta seção apresenta um exemplo que traz vários detalhes que devem ser levados em consideração dependendo do operador. O exemplo encapsula um vetor como uma classe com o objetivo de checar as indexações, evitando que posições aleatórias da memória sejam acessadas. Um vetor normal de C++ permite a indexação com qualquer inteiro; mesmo se o índice estiver além da área alocada o acesso é permitido, com conseqüências imprevisíveis.

O operador [ ] será redefinido para permitir o uso dos objetos desta classe da mesma maneira que um vetor C++. Além do operador, a classe terá um construtor que aloca os elementos do vetor. O funcionamento será o seguinte: o construtor, além de alocar os elementos, guarda o tamanho do vetor. O operador, que retorna o elemento correspondente do vetor alocado, checa se o índice é válido (o construtor guarda o tamanho) antes de fazer o acesso. Alguma coisa da forma:

```
float Vector::operator[](int i)
{
   if (i>=0 && i<size) return elems[i];
   else
   {
      printf("índice %d inválido\n", i);
      return -1.0;
   }
}
```

No entanto isto não é suficiente. É preciso lembrar que este operador pode ser usado de duas maneiras:

```
a = vetor[10];
vetor[20] = b;
```

Na primeira linha não há problema. O operador é uma função que retorna um valor, que será atribuído à variável *a*. Já na segunda linha a atribuição não é permitida, pois a função retorna o valor da posição 20 do vetor; para a atribuição é necessário saber o endereço da posição 20 do vetor. A solução é retornar uma referência para um float. Assim o valor de retorno é o endereço (necessário na segunda linha), mas este é

usado como um valor (primeira linha). Se simplesmente mudarmos o tipo de retorno da função para float&, um erro será sinalizado durante a sua compilação, pois o valor -1.0, retornado caso o índice seja inválido, não tem endereço. Ou seja, é necessário retornar uma variável, mesmo nesse caso. Outra particularidade deve ser observada aqui: como que será retornado será o endereço da variável retornada, esta precisa continuar existindo mesmo depois que afunção termina a sua execução. O que significa que não se pode retornar uma variável local, pois variáveis locais deixam de existir assim que a função termina. A implementação abaixo utiliza uma variável static dentro da função só para este fim:

```
class Vector {
   float *elems;
   int    size;
 public:
   Vector(int s);
   float& operator[](int i);
};

Vector::Vector(int s)
{
   size=s;
   elems = new float[size];
}

float& Vector::operator[](int i)
{
   if (i>=0 && i<size) return elems[i];
   else
   {
      static float lixo;
      printf("índice %d inválido\n", i);
      return lixo;
   }
}
```

Operadores de conversão

É possível definir operadores especiais para conversão de tipos. Além das conversões padrão, o programador pode definir como um objeto pode ser convertido para algum outro tipo. Consideremos uma classe *Arquivo* que modela um arquivo. Internamente esta classe pode ter um ponteiro para um arquivo (tipo FILE*) privado. Se fosse necessário fazer alguma operação já definida na biblioteca que não tenha sido mapeada na classe, seria preciso ter acesso ao ponteiro para arquivo. Ao invés de tornar público este campo, seria mais elegante definir como um objeto do tipo *Arquivo* se converte em um FILE*, que é o objetivo. Esta definição se dá da seguinte forma:

```
class Arquivo {
   FILE *file;
 public:
   Arquivo( char* nome ) { file=fopen(nome, "r"); }
   ~Arquivo()  { fclose(file); }
   char read() { return file?fgetc(file):EOF; }
   int aberto() { return file!=NULL; }
   operator FILE*() { return file; }
}

void main()
{
   int i;
   Arquivo arq("teste.c");
   fscanf( (FILE*)arq, "%d", &i );
}
```

Exercício 6 - Classe Complex com operadores globais

Alterar a classe Complex para permitir expressões do tipo 1+c, onde c é complexo.

Exercício 7 - Calculadora RPN para complexos

Modificar a calculadora para operar com números complexos.

Exercício 8 - Classe String

Implementar uma classe String. A classe deve permitir comparações, concatenações, atribuições e acesso a cada caracter, além de poder ser utilizada em funções como printf. O usuário desta classe não precisa se preocupar com tamanho de memória alocado. Ou seja, se for necessário realocar memória para uma atribuição ou uma concatenação, esta realocação deve ser feita automaticamente dentro da classe. O acesso aos caracteres deve ser seguro. Por exemplo, no caso de a string ter 10 caracteres e o usuário tentar escrever no vigésimo, isto não pode alterar uma área de memória aleatória.

Capítulo 5

Aspectos de reutilização

> *"Por que software não é como hardware? Por que todo desenvolvimento começa do nada? Deviam existir catálogos de módulos de software, assim como existem catálogos de chips: quando nós construímos um novo sistema, nós deveríamos estar usando os componentes destes catálogos e combinando-os, em vez de sempre reinventar a roda. Nós escreveríamos menos software, e talvez faríamos um desenvolvimento melhor . Será que alguns problemas dos quais todo mundo reclama - custos altos, prazos insuficientes, pouca confiabilidade - não desapareceriam? Por que não é assim?"*

Talvez você já tenha ouvido esta argumentação antes; talvez você próprio já tenha pensado nisso. Em 1968, no famoso workshop da OTAN sobre a crise de software, D. McIlroy já estava pregando a produção de componentes de software em massa. A reutilização, como um sonho, não é nova.

Qualquer pessoa que lide com o desenvolvimento de software se impressiona com seu caráter repetitivo. Diversas vezes, os programadores constroem funções e programas com o mesmo padrão: ordenação, busca de um elemento, comparação, etc. Uma maneira de interessante de avaliar esta situação é responder a seguinte pergunta: Quantas vezes, nos últimos seis meses, você escreveu uma rotina de busca de um elemento x em uma tabela t ?

As dificuldades técnicas de reutilização se tornam mais visíveis quando se observa a natureza das repetições no desenvolvimento de sistemas. Esta análise revela que apesar dos programadores tenderem a escrever os mesmos tipos de rotinas diversas vezes, estas não são exatamente iguais. Se fosse, a solução mais simples teoricamente; na prática, porém, muitos detalhes mudam de implementação para implementação (tipos dos elementos, estrutura de dados associada, etc.).

Apesar das dificuldades, algumas soluções foram propostas com relativos sucessos:

- Reutilização de código-fonte: Muito comum no meio científico. Muito da cultura UNIX foi difundida pelos laboratórios e universidades do mundo graças à disponibilidade de código-fonte ajudando estudadentes a estudarem, imitarem e estenderem o sistema. No entanto, esta não é a forma mais utilizada nos meios industrial e comercial além de que esta técnica não suporta ocultação de informação (information hiding).
- Reutilização de pessoal: É uma forma muito comum na indústria. Consiste na transferência de engenheiros de software de projetos a projetos fazendo a permanência de know-how na compania e assegurando a aplicação de experiências passadas em novos projetos. Obviamente, esta é uma maneira não-técnica e limitada.
- Reutilização de design: A idéia por trás desta técnica é que as companhias devem acumular repositórios de idéias descrevendo designs utilizados para os tipos de aplicação mais comuns.

Requisitos para reuso

As idéias apresentadas anteriormente, apesar de limitadas, mostram aspectos importantes para a reutilização de código:

A noção de reuso de código-fonte lembra que software é definido pelo seu código. Uma política satisfatória de reutilização deve produzir programas (códigos) reutilizáveis.

A reutilização de pessoal é fundamental pois os componentes de software são inúteis sem profissionais bem treinados e com experiência para reconhecer as situações com possibilidade de reuso.

A reutilização de design enfatiza a necessidade de componentes reutilizáveis que estejam em um nível conceitual e de generalidade alto — não somente com soluções para problemas específicos. Neste aspecto, poderá ser visto como o conceito de classes nas linguagens orientadas por objetos pode ser visto como módulos de design e de implementação.

A maneira de produzir módulos que permitem boa possibilidade de reuso é descrita abaixo. Neste caso, o nosso exemplo de procurar um elemento x em uma tabela t é bem ilustrativo.

Variação no tipo

Um módulo que implemente uma determinada funcionalidade deve ser capaz de fazê-lo sobre qualquer tipo a ele atribuído. Por exemplo, no caso da busca de um elemento x, o módulo deve ser aplicável a diferentes instâncias de tipo para x. É interessante a utilização do mesmo módulo para procurar um inteiro numa tabela de inteiros ou o registro de um empregado na sua tabela correspondente, etc.

Variação nas estruturas de dados e algoritmos

No caso da busca de um elemento x, o modo de busca pode ser adaptado para diversos tipos de estruturas de dados e algoritmos de busca associados: tabelas seqüenciais (ordenadas ou não), vetores, listas, árvores binárias, B-trees, diferentes estruturas de arquivos, etc. Neste sentido, o módulo deve suportar variações nas estruturas a ele associado.

Existência de rotinas relacionadas

De forma a fazer uma pesquisa em uma tabela, deve-se saber como esta é criada, como os elementos podem ser inseridos, retirados, etc. Deste modo, uma rotina de busca não é por si só suficiente; há a necessidade do acoplamento de diversas rotinas primitivas e relacionadas entre si.

Independência de representação

Uma estrutura modular verdadeiramente flexível habilita seus clientes uma operação sem o conhecimento de modo pelo qual o módulo foi implementado. Por exemplo, deve ser possível ao cliente escrever a seguinte chamada para a busca de x:

```
esta_presente = BUSCA( x, T );
```

sem saber qual o tipo da tabela T no momento da chamada. Se vários algoritmos de busca foram implementados, os mecanismos internos do módulo são responsáveis de saber qual o apropriado sem a intervenção do cliente. De maneira simplificada, isto é uma extensão do princípio de ocultação de informação pois havendo a necessidade de mudança na implementação, os clientes estão protegidos.

No entanto, a idéia vai mais além. O princípio da independência de representação não significa somente que mudanças na representação devem ser invisíveis para os clientes durante o ciclo de desenvolvimento do sistema: os clientes devem ser imunes também a mudanças durante a execução. No exemplo acima, é desejável que a rotina BUSCA se adapte automaticamente para a forma de T em tempo de execução mesmo que esta forma tenha sido alterada do instante da última chamada.

Este requisito é importante não somente pela questão do reuso mas também pela extensibilidade. Se T pode mudar de forma em tempo de execução, então uma decisão no sistema deve ser tomada para a utilização da versão de BUSCA correta. Em outras palavras, se não houver esse mecanismo automático o código, em algum local, deve conter um controle do tipo:

```
se T é do tipo A então “mecanismo A”
se T é do tipo B então “mecanismo B”
se T é do tipo C então “mecanismo C”
```

A estrutura de decisão deve estar ou no módulo ou no cliente. Ambos os casos não são satisfatórios. Se a decisão estiver no módulo, este módulo deve saber sobre todos as possibilidades existentes. Tal política pode levar a construção de módulos difíceis de gerenciar e sujeitos a constantes manutenções. Deixar a decisão para o cliente não é melhor. Desta forma, o cliente é obrigado a especificar que T é uma tabela do tipo A, B, etc. mas não é requerido a dizer mais nada: esta informação é suficiente para determinar que variante de BUSCA deve ser utilizada.

A solução para o problema é introduzida pelas linguagens orientadas por objetos com o mecanismo de herança em que o desenvolvimento é feito através da descentralização da arquitetura modular. Esta é construída por sucessivos incrementos e modificações conectadas por relações bem definidas que definem as versões corretas de BUSCA. Este mecanismo é chamado de Amarração Dinâmica (Late-Binding ou Dynamic binding).

Semelhanças nos subcasos

Este último item em reuso afeta o design e a construção dos módulos e não seus clientes. Este é fundamental pois determina a possibilidade da construção de módulos bem construídos sem repetições indesejáveis. O aparecimento de repetições excessivas nos módulos compromete suas consistências internas e torna-os difíceis de fazer manutenção.

O problema surge como os programadores podem se aproveitar e tomar vantagem de semelhanças em subcasos. Para tal, deve existir no conjunto de possibilidades de implementação subgrupos de soluções

com a mesma estrutura. No exemplo de busca na tabela, um exemplo típico é o aparecimento de implementações relacionadas com tabelas seqüenciais. Neste caso, o algoritmo pode ser descrito da mesma forma para todos os subcasos diferindo apenas em um conjunto reduzido de rotinas primitivas. A figura abaixo ilustra o algoritmo seqüencial genérico e algumas implementações de operações primitivas.

```
int BUSCA( Elemento x; TabelaSequencial T )
{
   int Pos
   COMEÇA();
   while not FINAL() && not ACHOU( pos, x, T ) do
      MOVE_PROXIMO();
   return not FINAL();
}
```

	Vetor	Lista Encadeada	Arquivo Seq.
COMEÇA()	i := 1	l := ponta_lista	rewind()
MOVE_PROXIMO()	i := i + 1	l := l.next	read()
FINAL()	i > tamanho	l == NULL	eof()

Neste caso, evita-se a repetição do método de BUSCA nas implementações de busca seqüencial. Tem-se uma única função de pesquisa que se difere em quais funçoes específicas de COMEÇA, MOVE_PRÓXIMO e FINAL serão chamadas.

Os mecanismos decritos acima são compreendidos nas linguagens orientadas por objetos pelo mecanismo de herança descrito a seguir.

Herança

Provavelmente herança é o recurso que torna o conceito de classe mais poderoso. Em C++, o termo herança se aplica apenas às classes. Variáveis não podem herdar de outras variáveis e funções não podem herdar de outras funções.

Herança permite que se construa e estenda continuamente classes desenvolvidas por você mesmo ou por outras pessoas, sem nenhum limite. Começando da classe mais simples, pode-se derivar classes cada vez mais complexas que não são apenas mais fáceis de debuggar, mas elas próprias são mais simples.

O objetivo de um projeto em C++ é desenvolver classes que resolvam um determinado problema. Estas classes são geralmente construídas incrementalmente começando de uma classe básica simples, através de herança. Cada vez que se deriva uma nova classe começando de uma já existente, pode-se herdar algumas ou todas as características da classe pai, adicionando novas quando for necessário. Um projeto completo pode ter centenas de classes, mas normalmente estas classes são derivadas de algumas poucas classes básicas. C++ permite não apenas herança simples, mas também múltipla, permitindo que uma classe incorpore comportamentos de todas as suas classes bases.

Reutilização em C++ se dá através do uso de uma classe já existente ou da construção de uma nova classe a partir de uma já existente.

Classes derivadas

A descrição anterior pode ser interessante, mas um exemplo é a melhor forma de mostrar o que é herança e como ela funciona. Aqui está um exemplo de duas classes, a segunda herdando as propriedades da primeira:

```
class Caixa {
 public:
   int altura, largura;
   void Altura(int a)  { altura=a; }
   void Largura(int l) { largura=l; }
};

class CaixaColorida : public Caixa {
 public:
   int cor;
   void Cor(int c) { cor=c; }
};
```

Usando a terminologia de C++, a classe Caixa é chamada classe base para a classe *CaixaColorida*, que é chamada classe derivada. Classes base são também chamadas de classes pai. A classe *CaixaColorida* foi declarada com apenas uma função, mas ela herda duas funções e duas variáveis da classe base. Sendo assim, o seguinte código é possível:

```
void main()
{
   CaixaColorida cc;
   cc.Cor(5);
   cc.Largura(3);  // herdada
   cc.Altura(50);  // herdada
}
```

Note que as funções herdadas são udadas exatamente como as não herdadas. A classe Colorida não precisou sequer mencionar o fato de que as funções *Caixa::Altura()* e *Caixa::Largura()* foram herdadas. Esta uniformidade de expressão é um grande recurso de C++. Usar um recurso de uma classe não requer que se saiba se este recurso foi herdado ou não, já que a notação é invariante. Em muitas classes pode existir uma cadeia de classes base derivadas de outras classes base. Uma classe herdada de uma árvore de herança como esta herdaria características de muitas classes pai diferentes. Entretanto, em C++, não é preciso se preocupar onde ou quando um recurso foi introduzido na árvore.

Derivar uma classe de outra aumenta a flexibilidade a um custo baixo. Uma vez que já existe uma classe base sólida, apenas as mudanças feitas nas classes derivadas precisam ser depuradas. Mas quando exatamente se usa uma classe base, e que tipos de modificações precisam ser feitas? Quando se herda característias de uma classe base, a classe derivada pode estender, restringir, modificar, eliminar ou usar qualquer dos recursos sem qualquer modificação.

O que não é herdado

Nem tudo é herdado quando se declara uma classe derivada. Alguns casos são inconsistentes com herança por definição:

- Construtores
- Destrutores
- Operadores new
- Operadores de atribuição (=)
- Relacionamentos friend
- Atributos privados

Classes derivadas invocam o construtor da classe base automaticamente, assim que são instanciadas.

Membros de classes protected

Na seção de controle de acesso, vimos como deixar disponíveis ou ocultar atributos das classes, usando os especificadores public e private. Além desses dois, existe um outro especificador, protected. Do ponto de vista de fora da classe, um atributo protected funciona como private: não é acessível fora da classe; a diferença está na herança. Enquanto um atributo private de uma classe base não é visível na classe derivada, um protected é, e continua sendo protected na classe derivada. Por exemplo:

```
class A {
 private:
   int a;
 protected:
   int b;
 public:
   int c;
};

class B : public A {
 public:
   int geta() { return a; }  // ERRO!! a não é visível
   int getb() { return b; }  // válido (b protected)
   int getc() { return c; }  // válido (c public)
};

void main()
{
   A ca;
   B cb;

   ca.a = 1; // ERRO! a não é visível (private)
   ca.b = 2; // ERRO! b não é visível de fora (protected)
   ca.c = 3; // válido (c é public)
```

```
   cb.a = 4; // ERRO! a não é visível nem internamente em B
   cb.b = 5; // ERRO! b continua protected em B
   cb.c = 6; // válido (c continua public em B)
}
```

Construtores e destrutores

Quando uma classe é instanciada, seu construtor é chamado. Se a classe foi derivada de alguma outra, o construtor da classe base também precisa ser chamado. A ordem de chamada dos construtores é fixa em C++. Primeiro a classe base é construída, para depois a derivada ser construída. Se a classe base também deriva de alguma outra, o processo se repete recursivamente até que uma classe não derivada é alcançada.

Desta forma, quando um construtor para uma classe derivada é chamado, todos os procedimentos efetuados pelo construtor da classe base já foram realizados. Considere a seguinte árvore de herança:

```
class Primeira {};
class Segunda: public Primeira {};
class Terceira: public Segunda {};
```

Quando a classe Terceira é instanciada, os construtores são chamados da seguinte maneira:

```
Primeira::Primeira();
Segunda::Segunda();
Terceira::Terceira();
```

Esta ordem faz sentido, já que uma classe derivada é uma especialização de uma classe mais genérica. Isto significa que o construtor de uma classe derivada pode usar atributos herdados.

Os destrutores são chamados na ordem inversa dos construtores. Primeiro, os atributos mais especializados são destruídos, depois os mais gerais. Então a ordem de chamada dos destrutores quando Terceira sai do escopo é:

```
Terceira::~Terceira();
Segunda::~Segunda();
Primeira::~Primeira();
```

Como os construtores das classes base são chamados automaticamente, deve existir alguma maneira de passar os argumentos corretos para estes construtores, no caso de eles necessitarem de parâmetros. Existe uma notação especial para este caso, ilustrada abaixo, para funções inline e não inline:

```
class Primeira {
   int a, b, c;
 public:
   Primeira(int x, int y, int z) { a=x; b=y; c=z; }
};

class Segunda : public Primeira {
   int valor;
 public:
   Segunda(int d) : Primeira(d, d+1, d+2) { valor = d; }
   Segunda(int d, int e);
};

Segunda::Segunda(int d, int e) : Primeira(d, e, 13)
{
   valor = d + e;
}
```

A partir do exemplo acima, não é difícil perceber que, se uma classe base não possui um construtor sem parâmetros, a classe derivada tem que, obrigatoriamente, declarar um construtor, mesmo que este construtor seja vazio:

```
class Base {
 protected:
   int valor;
 public:
   Base(int a) { valor = a; }
   // esta classe não possui um construtor
   // sem parâmetros
};

class DerivadaErrada : public Base{
 public:
   int pegaValor() { return valor; }
```

```
    // ERRO! classe não declarou construtor, compilador não
    // sabe que parâmetro passar para Base
};

class DerivadaCerta: public Base {
 public:
    int pegaValor() { return valor; }
    DerivadaCerta() : Base(0) {}
    // CERTO: mesmo que não haja nada a fazer
    // para inicializar a classe,
    // é necessário declarar um construtor
    // para dizer com que parâmetro
    // construir a classe Base
};
```

Herança pública x herança privada

Nos exemplos acima, em toda declaração de uma classe derivada, usou-se a palavra public:

```
class B : public A { ...
```

Na realidade, os especificadores de acesso private e public podem ser usados na declaração de uma herança. Por default, as heranças são private, por isso usou-se public nos exemplos acima.

Estes especificadores afetam o nível de acesso que os atributos terão na classe derivada:

- private: todos os atributos herdados (public, protected) tornam-se private na classe derivada;
- public: todos os atributos public são public na classe derivada, e todos os protected também continuam protected.

Na realidade, isto é uma consequência da finalidade real de heranças public e protected, que voltará a ser discutida em compatibilidade de tipos.

Exercício 9 - Calculadora como um objeto. Classe RPN.

Alterar a calculadora para transformar a própria calculadora em um objeto

Capítulo 6

Polimorfismo

A origem da palavra polimorfismo vem do grego: poli (muitos) e morphos (forma) - múltiplas formas. Polimorfismo descreve a capacidade de um código C++ se comportar de diferentes formas dependendo do contexto em tempo de execução.

Este é um dos recursos mais poderosos de linguagens orientadas a objetos (se não o mais), que permite trabalhar em um nível de abstração bem alto ao mesmo tempo que facilita a incorporação de novos pedaços em um sistema já existente. Em C++ o polimorfismo se dá através da conversão de ponteiros (ou referências) para objetos.

Conversão de ponteiros

Normalmente se usam não objetos de classes isoladas, mas sim objetos em uma hierarquia de classes. Considere as seguintes classes:

```
class A {
   public: void f();
};

class B: public A {
   public: void g();
};
```

Como *B* é derivado de *A*, todos os membros disponíveis em *A* (função *f*) também estarão disponíveis em *B*. Então *B* é um superconjunto de *A*, e todas as operações que podem ser feitas com objetos da classe *A* também podem ser feitas com objetos do tipo *B*. A classe *B* é uma especialização da classe *A*, e é não só um objeto do tipo *B*, mas também um objeto do tipo *A*. Nada impede que objetos da classe *B* sejam vistos como sendo da classe *A*, pois todas as operações válidas para *A* são também válidas para *B*. Ver um objeto do tipo *B* como sendo do tipo A significa convertê-lo para o tipo *A*. Esta conversão pode ser feita, sempre no sentido da classe mais especializada para a mais básica. *A* conversão inversa não é permitida, pois operações específicas de *B* não são válidas sobre objetos da classe *A*. Conversão aqui não deve ser entendida como cópia. A simples atribuição de um objeto do tipo *B* para um objeto do tipo *A* copia a parte *A* do objeto do tipo *B* para o objeto do tipo *A*. O polimorfismo é feito através da conversão de ponteiros. O exemplo abaixo mostra as várias alternativas:

```
void main()
{
   A a, *pa; // pa pode apontar para objetos do tipo A e derivados
   B b, *pb; // pb pode apontar para objetos do tipo B e derivados

   a = b;   // copia a parte A de b para a (não é conversão)
   b = a;   // erro! a pode não ter todos elementos para a cópia
   pa = &a; // ok
   pa = &b; // ok, pa aponta para um objeto do tipo B
   pb = pa; // erro! pa pode apontar para um objeto do tipo A
   pb = &b; // ok
   pb = &a; // erro! pb não pode apontar para objetos do tipo A
}
```

Para tirar qualquer dúvida sobre quais conversões podem ser feitas, o exemplo abaixo mostra o que pode ser feito com este recurso a partir das classes *A* e *B*:

```
void chamaf(A* a) // pode ser chamada para A e derivados
{
   a->f();
}

void chamag(B* b) // pode ser chamada para B e derivados
{
   b->g();
}
```

```
void main()
{
   A a;
   B b;
   chamaf(&a); // ok, a tem a função f
   chamag(&a); // erro! a não tem a função g
               // (a não pode ser convertido para o tipo B)
   chamaf(&b); // ok, b tem a função f
   chamag(&b); // ok, b tem a função g
}
```

Repare que as funções *chamaf* e *chamag* foram escritas para os tipos *A* e *B*, mas podem ser usadas com qualquer objeto que seja derivado destes. Se um novo objeto derivado de *B* for criado no futuro, a mesma função poderá ser usada sem necessidade de recompilação.

Estas conversões só podem ser feitas quando a herança é pública. Se a herança for privada a conversão não é permitida.

Redefinição de métodos em uma hierarquia

Não existe sobrecarga em uma hierarquia. A definição de um método com mesmo nome de uma classe básica não deixa os dois disponíveis, mesmo que os tipos dos parâmetros sejam diferentes. Os métodos da classe básica de mesmo nome são escondidos. Eles não ficam inacessíveis, mas não podem ser chamados diretamente:

```
class A {
   public: void f();
};

class B : public A {
  public:
   void f(int a);     // f(int) esconde f()
   void f(char* str);
};

void main()
{
   B b;
   b.f(10);     // ok, função f(int) de B
   b.f("abc");  // ok, função f(char*) de B
   b.f();       // erro! f(int) escondeu f()
   b.A::f();    // ok
}
```

É possível também declarar um método com mesmos nome e assinatura (tipo de retorno e tipo dos parâmetros) que um da classe base. O novo método esconde o da classe base, que precisa do operador de escopo para ser acessado. No entanto, esta redefinição merece atenção especial. Considerando o exemplo:

```
class A {
 public: void f();
};
class B : public A {
 public: void f();
};

void chamaf(A* a) { a->f(); }

void main()
{
   B b;
   chamaf(&b);
}
```

A função *chamaf* pode ser usada para qualquer objeto do tipo *A* e derivados. No exemplo acima, ela é chamada com um objeto do tipo *B*. O método *f* é chamado no corpo de *chamaf*. Mas qual versão será executada? No exemplo acima o método executado será *A::f*.

Isto é o que acontece, mas será que este é o comportamento desejado? Se o polimorfismo nesse caso for encarado como uma maneira diferente (mais limitada) de ver o mesmo objeto, não seria natural

chamar o método *B::f*? Afinal, o objeto é do tipo *B*, apenas está "guardado" em um ponteiro para o tipo *A*. O comportamento ideal pode não estar claro agora. A seção seguinte procura esclarecer este ponto.

Exemplo: classes List e ListN

A classe abaixo implementa uma lista encadeada:

```
class List {
 public:
   List();
   int add(int);    // retorna 0 em caso de erro, 1 ok
   int remove(int); // retorna 0 em caso de erro, 1 ok
};
```

Suponha que esta declaração é parte de uma biblioteca cujo código fonte não está disponível. Ou seja, a declaração acima faz parte de um header file a ser incluído em arquivos que utilizem a biblioteca.

Através do mecanismo de herança, é possível criar uma nova lista que retorna o número de elementos contidos na lista:

```
class ListN : public List {
   int n;
 public:
   ListN() { n=0; }
   int nelems() { return n; }
   int add(int i)
   {
      int r = List::add(i);
      if (r) n++;
      return r;
   }
   int remove(int i)
   {
      int r = List::remove(i);
      if (r) n--;
      return r;
   }
};
```

A nova classe foi criada sem nenhum conhecimento sobre a implementação da classe *List*.

Para completar o exemplo, falta utilizar as declarações acima:

```
void manipula_lista(List* l)
{
   // insere e remove vários elementos na lista
}

void main()
{
   List  l1;
   ListN l2;
   manipula_lista(&l1);
   manipula_lista(&l2);
   printf("a lista l2 contém %d elementos\n", l2.nelems());
}
```

A função *manipula_lista* utiliza apenas os métodos *add* e *remove*, portanto pode operar tanto sobre objetos do tipo *List* quanto do tipo *ListN*. Supondo que esta função retorna deixando cinco elementos na lista, qual será o resultado do printf? Assim como na seção anterior, as funções chamadas em *manipula_lista* serão *List::add* e *List::remove*. Como estas funções não alteram a variável *n*, o resultado do printf será:

```
a lista l2 contém 0 elementos
```

Ou seja, a manipulação deixou o objeto *l2* inconsistente internamente. Para manter a sua consistência, seria preciso que os métodos *ListN::add* e *ListN::remove* fossem chamados, o que não está acontecendo.

Early x late binding

Nos exemplos acima está acontecendo o que se chama de early-binding. Early-binding é a ligação dos identificadores em tempo de compilação. Quando a função *manipula_lista* foi compilada, o código gerado tem uma chamada explícita à função *A::f*. Com isso, qualquer que seja o objeto passado como parâmetro,

a função chamada será sempre a mesma. Este é o processo de ligação utilizado em linguagens de programação convencionais, como C e Pascal.

O problema de early-binding é que o programador precisa saber que objetos serão usados em todas as chamadas de função em todas as situações. Isto é uma limitação. A vantagem é a eficiência; a chamada à função é feita diretamente, pois o código gerado tem o endereço físico da mesma.

Late-binding é um tipo de ligação que deixa a amarração dos nomes para ser feita durante a execução do programa. Isto é, dependendo da situação a amarração pode ser feita a funções diferentes durante a execução do programa. É exatamente o que falta no exemplo da lista encadeada. A função *manipula_lista* deve chamar ora *List::add* ora *ListN:add*, dependendo do tipo do objeto.

O problema de late-binding é exatamente a eficiência. O código deve descobrir que função chamar durante a execução do programa. Linguagens orientadas a objetos puras, como Smalltalk, usam exclusivamente late- binding. O resultado é uma linguagem extremamente poderosa, mas com algumas penalidades em relação ao tempo de execução. Por outro lado, ANSI C usa somente early-binding, resultando em alta velocidade mas falta de flexibilidade.

C++ não é uma linguagem procedural tradicional como Pascal, mas também não é uma linguagem orientada a objetos pura. C++ é uma linguagem híbrida. C++ usa early e late binding procurando oferecer o melhor de cada um dos métodos. O programador controla quando usar um ou outro. Para um código em que a execução é determinística, pode-se forçar C++ para usar early-binding. Em situações mais complexas, usa-se late-binding. Desta forma, pode-se conciliar alta velocidade com flexibilidade.

Métodos virtuais

Em C++, late-binding é especificado declarando-se um método como virtual. Late-binding só faz sentido para objetos que fazem parte de uma hierarquia de classes. Se um método *f* é declarado virtual em uma classe *Base* e redefinido na classe *Derivada*, qualquer chamada a *f* sobre um objeto do tipo *Derivada*, mesmo que via um ponteiro para *Base*, executará *Derivada::f*. A redefinição de um método virtual é também virtual. A especificação virtual nesse caso é redundante.

Este mecanismo pode ser usado com as listas encadeadas para manter a consistência dos objetos do tipo *ListN*. Basta declarar, na classe *List*, os métodos *add* e *remove* como virtuais:

```
class List {
 public:
   List();
   virtual int add(int);    // retorna 0 em caso de erro, 1 ok
   virtual int remove(int); // retorna 0 em caso de erro, 1 ok
};
```

Agora a função *manipula_lista* executará as versões corretas de *add* e *remove*.

Destrutores virtuais

Não faz sentido construtores poderem ser virtuais, já que eles não são chamados a partir de um objeto, mas sim para criar objetos. Destrutores, apesar de serem métodos especiais, podem ser virtuais porque são chamados a partir de um objeto. A chamada se dá em duas situações: quando o objeto sai do escopo e quando ele é destruído com delete. Na primeira situação, o compilador sabe o tipo exato do objeto e portanto chama o destrutor correto. Já na segunda situação isso pode não ocorrer, como mostra o exemplo abaixo:

```
class A {
   // ...
};

class B : public A {
   int* p;
 public:
   B(int size) { p = new int[size]; }
   ~B() { delete [] p; }
   // ...
};

void destroy(A* a)
{
   delete a; // chama destrutor
}

void main()
```

```
{
   B* b = new B(20);
   destroy(b);
}
```

O que está acontecendo é exatamante o que acontecia na primeira versão das listas encadeadas. A função *destroy* chama (com early-binding) diretamente o destrutor *A::~A*, quando o certo seria chamar *B::~B* antes, para depois chamar *A::~A*. Para forçar este funcionamento correto, é preciso que o destrutor seja virtual. Caso contrário, objetos podem ser destruídos e deixando alguma coisa para trás.

Tabelas virtuais

Esta seção tem por objetivo esclarecer um pouco o que acontece por trás dos métodos virtuais, ou seja, como late-binding é implementado. Este conhecimento ajuda o entendimento dos métodos virtuais, suas limitações, poderios e eficiência.

Imaginando a seguinte hierarquia de classes:

```
class A {
   int a;
 public:
   virtual void f();
   virtual void g();
};
class B : public A {
   int b;
 public:
   virtual void f(); // redefinição de A::f
   virtual void h();
};
void chamaf(A *a) { a->f(); }
```

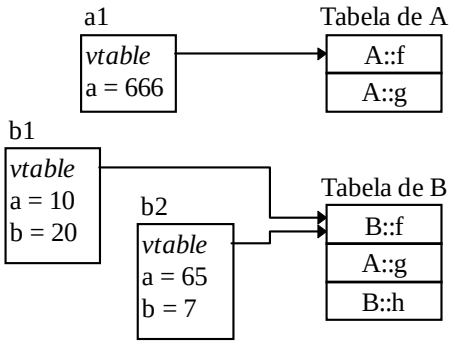
A função *chamaf* executará o método *f* do objeto passado como parâmetro. Dependendo do tipo do objeto, a mesma linha executará *A::f* ou *B::f*. Ou seja, é como se a função *chamaf* fosse implementada internamente assim:

```
void chamaf(A *a)
{
   switch (tipo de a)
   {
      case A: a->A::f(); break;
      case B: a->B::f(); break;
   }
}
```

Mas isto significa que cada classe derivada de *A* precisa de um case neste switch, o que tem várias desvantagens: esta função não poderia ser utilizada com novas classes que venham a ser criadas no futuro; quanto mais classes na hierarquia pior seria a eficiência e, acima de tudo, o compilador não tem como saber todas as classes derivadas de *A*. Esta análise mostra que esta implementação é irreal, e outra estratégia deve ser usada.

Para resolver este problema, C++ utiliza tabelas virtuais, que são descrições dos métodos de uma determinada classe. Estas tabelas são vetores de funções. O número de entradas da tabela é igual ao número de métodos virtuais da classe e cada posição da tabela tem o ponteiro para uma função virtual. Quando uma classe tem algum método virtual, todos os objetos desta classe terão uma referência para esta tabela, que é o descritor da classe.

No exemplo acima, existem duas tabelas virtuais: uma para a classe *A* e outra para *B*. Quando um objeto é criado, ele leva uma referência para esta tabela. A tabela de *A* tem duas posições, uma para cada método virtual (*f* e *g*). A tabela de *B* tem uma posição a mais para o método *h*. A posição dos métodos na tabela é sempre a mesma, ou seja, se na tabela de *A* a primeira posição apontar para o método *f*, em todas as classes derivadas a primeira posição será de *f*. Na tabela de *A*, este ponteiro aponta para *A::f*, enquanto que em *B* ele aponta para *B::f*. Quando acontece alguma chamada no código, a função não é chamada pelo nome, e sim por indexação a esta tabela. Em qualquer tabela de classes derivadas de *A* o método *f* estará na mesma posição, no caso, a primeira. A figura abaixo mostra as possíveis tabelas e objetos para o exemplo de *A* e *B* com alguns objetos na memória:



Considerando que a tabela virtual é um campo de todos objetos (com nome *vtable* por exemplo), a função *chamaf* pode ser implementada assim:

```
void chamaf(A *a)
{
   a->vtable[0](); // posição 0 corresponde ao método f
}
```

Esta implementação funcionará com qualquer objeto derivado de *A*, até mesmo de classes futuras, pois não exige que o compilador saiba quais são as classes existentes. A eficiência é sempre a mesma independente de quantas classes existam ou qual a sua altura na árvore de hierarquia. Esta eficiência não é a mesma de uma chamada de função normal (com early-binding) pois envolve uma indireção.

Classes abstratas - Métodos virtuais nulos

O mecanismo de polimorfismo apresentado até aqui permite a especialização de objetos mantendo sua consistência e sem invalidar códigos que manipulem objetos mais básicos. Esta última característica permite a construção de um sistema inteiro a partir de poucas classes básicas, independente das especializações que vierem a ser feitas.

Se um sistema exige uma pilha por exemplo, é possível implementar uma limitada (com vetores por exemplo) e contruir todo o sistema em cima desta. Depois do sistema pronto, pode-se alterar a pilha para uma implementação mais sofisticada (listas encadeadas) sem necessidade de alterar todo o código que manipula a pilha. Basta que a nova pilha seja uma subclasse da pilha original.

Na realidade nem é necessário que a pilha original seja implementada, já que no futuro ela será substituída. Com os recursos vistos até aqui, deve haver uma implementação para todos os métodos da classe, mas C++ permite a declaração de classes sem implementação de alguns métodos. Métodos sem implementação são sempre virtuais e são chamados virtuais puros. Uma classe que tem pelo menos um método virtual puro é chamada de abstrata.

Não é permitido criar objetos de classes abstratas. Estas são feitas apenas para a criação de outras por herança. Classes derivadas de classes abstratas continuam sendo abstratas, a não ser que forneçam implementação para todos os métodos virtuais puros.

Classes abstratas estão em um nível intermediário entre especificação e programa. Uma classe abstrata é quase uma especificação; ao mesmo tempo é um elemento da linguagem de programação. Estas classes permitem a definição das interfaces dos objetos sem entrar em detalhes de implementação. A partir desta descrição já é possível implementar programas que manipulem estas classes pelo mecanismo de polimorfismo.

Uma aplicação que use uma pilha pode ser totalmente implementada a partir de uma classe abstrata que descreva uma pilha:

```
class Stack {
 public:
   virtual void push(int) = 0;
   virtual int  pop() = 0;
   virtual int  empty() = 0;
};
```

Apenas com esta declaração já é possível manipular pilhas. Qualquer objeto derivado deste pode ser usado onde se espera um *Stack*. A única restrição é a criação das pilhas. Para criar uma pilha real, é preciso declarar uma classe não abstrata derivada de *Stack*. Não abstrata significa com implementação dos três métodos de *Stack*.

Herança de tipo x herança de código

Vamos supor que é chegada a hora de implementar a pilha a ser usada na aplicação da seção anterior. Para que os objetos da nova classe possam ser utilizados na a aplicação, a pilha deve herdar de *Stack*:

```
class StackVector : public Stack { ... };
```

Agora vamos imaginar outra situação. É preciso implementar uma classe *Queue* (fila). Para facilitar esta tarefa, será usada uma classe *LinkedList*, que implementa uma lista encadeada. A especificação da fila inclui apenas os métodos *insert*, *remove* e *empty* (os métodos adicionais aos que são reutilizados da classe *LinkedList*). A declaração da classe fica assim:

```
class Queue: private LinkedList {
 public:
   void insert(int i);
   int  remove();
   int  empty();
};
```

Repare que, para a criação da fila, a herança usada foi a privada. Isto evita que os métodos da lista encadeada possam ser usados diretamente sobre a fila, o que seria um furo na consistência do objeto.

Estas duas situações têm mais diferenças do que uma simples herança pública ou privada. Conceitualmente estas duas heranças estão sendo feitas com objetivos totalmente diferentes. No primeiro caso, o objetivo da herança é simplesmente permitir a utilização da classe *StackVector* como uma *Stack*. Ou seja, aplicações que manipulem *Stack*s também podem ser usadas para manipular *StackVector*s. Nesse caso a herança só é feita para que a classe derivada tenha o tipo da classe base. É a herança de tipo.

No segundo caso, o objetivo não é usar objetos do tipo *Queue* em códigos que manipulem *LinkedList*. A herança está sendo usada para reutilizar o código escrito para implementar listas encadeadas, evitando a re-implementação do que já está pronto. É a herança de código.

Normalmente a herança privada se aplica a uma herança de código, enquanto que a pública se aplica à herança de tipo. Esta diferenciação é importante na hora de projetar sistemas.

Exemplo: árvore binária para qualquer objeto

Para exemplificar um uso de classes abstratas, esta seção implementa uma árvore binária simples. A primeira versão armazena floats na sua estrutura:

```
class BinTree {
   struct elem {
      elem* right;
      elem* left;
      float val;
      elem(float f) { val=f; right=left=0; }
   } *root;
   int look(elem* no, float f)
   {
      if (!no) return 0;
      if (no->val == f) return 1;
      if (no->val > f) return look(no->left, f);
      return look(no->right, f);
   }
   void put(elem*& no, float f)
   {
      if (!no) no = new elem(f);
      else if (no->val > f) put(no->left, f);
      else put(no->right, f);
   }
 public:
   BinTree() { root = 0; }
   int find(float v) { return look(root, v); }
   void insert(float v) { put(root, v); }
};
```

Analisando esta classe, chega-se à conclusão de que a árvore binária em si tem sempre o mesmo comportamento independente do tipo de dado que ela armazena. O tipo float só aparece porque este tipo tem que ser declarado. Se fosse necessária uma árvore binária para armazenar inteiros, a implementação seria exatamente a mesma. A única modificação necessária seria a substituição de todas as palavras float por int.

Com o uso de classes abstratas é possível fazer uma única implementação da árvore binária servir para vários tipos de dados. A idéia é criar uma classe abstrata que represente o tipo de dado a ser armazenado. Qualquer classe derivada desta pode ser usada com a árvore. A implementação da árvore passa a manipular objetos desta classe abstrata. Analisando a implementação acima, nota-se que, para que um tipo de dado seja incluído na árvore binária, ele precisa ter as operações de comparação > e ==. Isto define a classe abstrata:

```
class BinTreeObj {
 public:
   virtual int operator==(BinTreeObj&) = 0;
   virtual int operator> (BinTreeObj&) = 0;
};
```

A declaração destes métodos virtuais puros implica que qualquer tipo derivado deste precisa fornecer uma implementação para estes métodos para deixar de ser abstrato. A versão modificada da árvore fica assim:

```
class BinTree {
   struct elem {
      elem* right;
      elem* left;
      BinTreeObj& val;
      elem(BinTreeObj& f) : val(f) { right=left=0; }
   } *root;
   int look(elem* no, BinTreeObj& f)
   {
      if (!no) return 0;
      if (no->val == f) return 1;
      if (no->val > f) return look(no->left, f);
      return look(no->right, f);
   }
   void put(elem*& no, BinTreeObj& f)
   {
      if (!no) no = new elem(f);
      else if (no->val > f) put(no->left, f);
      else put(no->right, f);
   }
 public:
   BinTree() { root = 0; }
   int find(BinTreeObj& v) { return look(root, v); }
   void insert(BinTreeObj& v) { put(root, v); }
};
```

Como o polimorfismo se dá pela conversão de ponteiros, a árvore deve armazenar apenas ponteiros ou referências. Para utilizar esta árvore, basta fazer com que o tipo dos objetos a serem armazenados herdem da classe *BinTreeObj*. Para armazenar algum tipo primitivo é necessário criar uma nova classe derivada de *BinTreeObj* que armazene este tipo.

Conversão de um objeto básico para um derivado

Aproveitando a árvore binária da seção anterior, uma classe *String* será criada com o objetivo de armazenar o tipo char* na árvore. A classe deve herdar de *BinTreeObj* e declarar os dois métodos de comparação. Algo do tipo:

```
class String : public BinTreeObj {
   char *str;
 public:
   String(char* s) { str = s; }
   int operator==(String& o)
   { return !strcmp(str, o.str); }
   int operator> (String& o)
   { return  strcmp(str, o.str) > 0; }
};
```

No entanto, com a declaração acima, o compilador reclama que a classe *String* continua abstrata quando algum objeto é criado:

```
void main()
{
   BinTree bt;
```

```
   bt.insert( * new String("abc") ); // ERRO! String abstrata
}
```

O problema é que os métodos declarados em *String* não estão redefinindo os da classe abstrata, mas escondendo-os. Analisando a assinatura dos dois métodos, percebe-se a diferença:

```
int BinTreeObj::operator==(BinTreeObj&);
int String    ::operator==(String& o)
```

Se fosse o método de *String* estivesse redefinindo o de *BinTreeObj*, seria possível converter objetos de uma classe mais básica para uma mais específica. Como foi visto com polimorfismo, esta não é uma conversão segura. Na implementação da árvore binária, este método é chamado e o parâmetro é sempre uma referência para um *BinTreeObj*, que pode não ser uma *String*. Para redefinir um método, os parâmetros devem ser ou do mesmo tipo do declarado na classe base ou de um tipo mais básico. Nunca de um tipo derivado. O contrário acontece com o tipo de retorno de uma função.

Chegando à conclusão de que os parâmetros dos métodos de *String* devem ser do tipo *BinTreeObj*, surge outro problema. Como acessar o campo *str* e fazer a comparação das *String*s? Deve haver alguma maneira de converter um *BinTreeObj* em uma *String*, mesmo que esta conversão não seja segura. Na realidade esta conversão pode ser forçda através de type casts explícitos, assim como em C é possível converter um ponteiro para outro de um tipo completamente diferente. Se o objeto for realmente do tipo desejado, a conversão é feita. Mas se o tipo não for o esperado, os resultados são imprevisíveis. Mas nesse caso, é a única opção:

```
class String : public BinTreeObj {
   char *str;
 public:
   String(char* s) { str = s; }
   int operator==(BinTreeObj& o)
   { return !strcmp(str, ((String&)o).str); }
   int operator> (BinTreeObj& o)
   { return  strcmp(str, ((String&)o).str) > 0; }
};
```

Esta com certeza não é a melhor maneira de se implementar uma classe que manipula um tipo qualquer. Existem duas outras maneiras seguras, vistas mais à frente: templates e type casts dinâmicos.

Herança múltipla

Em C++, a herança não se limita a uma única classe base. Uma classe pode ter vários pais, herdando características de todos eles. Este tipo de herança introduz grande dose de complexidade na linguagem e no compilador, mas os benefícios são substanciais. Considere a criação de uma classe MesaRedonda, tendo não só propriedades de mesas, mas também as características geométricas de ser redonda. O código abaixo é uma possível implementação:

```
class Circulo {
   float raio;
 public:
   Circulo(float r) { raio = r; }
   float area()     { return raio*raio*3.14159; }
};

class Mesa {
   float ipeso;
   float ialtura;
 public:
   Mesa(float p, float a)  { ipeso = p; ialtura=a; }
   float peso()   { return ipeso; }
   float altura() { return ialtura; }
};

class MesaRedonda: public Circulo, public Mesa {
   int icor;
 public:
   MesaRedonda(int c, float a, float p, float r);
   int cor() { return icor; }
};
```

```
MesaRedonda::MesaRedonda(int c, float a, float p, float r)
 : Mesa(p, a), Circulo(r)
{
   icor = c;
}

void main()
{
   MesaRedonda mesa(5, 1, 20, 3.5 );
   printf("Peso:   %f\n", mesa.peso());
   printf("Altura: %f\n", mesa.altura());
   printf("Area:   %f\n", mesa.area());
   printf("Cor:    %d\n", mesa.cor());
};
```

Um exemplo natural poderia sair da seção que discute herança de tipo ou código. Para implementar uma pilha com listas encadeadas que possa ser usada como *Stack* e aproveitando uma classe já implementada de listas encadeadas, a declaração seria assim:

```
class StackList : public Stack, private LinkedList {
   // ...
};
```

Ordem de chamada dos construtores e destrutores

Assim como em herança simples, os construtores das classes base são chamados antes do construtor da classe derivada. A ordem de declaração na classe define a ordem de chamada dos construtores. No exemplo acima, a classe foi declarada com uma ordem

```
class MesaRedonda: public Circulo, public Mesa {
```

e o construtor com outra:

```
MesaRedonda::MesaRedonda(int c, float a, float p, float r)
: Mesa(p, a), Circulo(r)
```

Como a ordem de declaração na classe é a que define, a ordem dos construtores será:

```
Circulo::Circulo
Mesa::Mesa
MesaRedonda::MesaRedonda
```

Classes básicas virtuais

Classes base virtuais só são usadas com herança múltipla. É uma maneira de o programador controlar como as classes devem ser herdadas. Por exemplo:

```
class A {
 public:
   int a;
};
class B: public A {};
class C: public A{};
class D: public B, public C {
public:
   int valor() { return a; }
};
```

Este código gera uma hierarquia onde a classe D tem duas cópias da parte *A*, uma associada a *B* e outra a *C*. Portanto, o código acima gera um erro de compilação:

```
Member is ambiguous: 'A::a' and 'A::a'
```

O problema é que a declaração de *B* herdando de *A* faz com que a classe *B* já tenha uma "parte *A*" incorporada. Nesse caso, *B* já tem a sua variável *a*. O mesmo acontece com *C*. O resultado são duas cópias de *A* na hierarquia. Uma é a "parte *A*"de *B* e outra é a "parte *A"de C. O* compilador não sabe que cópia de *a* esta sendo referenciada. O operador de escopo poderia ser utilizado para retirar o erro:

```
int valor() { return C::a; }
```

Às vezes o programador quer montar uma árvore onde só exista uma cópia de *A*. É o caso de usar uma classe base virtual. Declarando uma classe base como virtual faz com que a classe derivada não inclua a classe base. Seria o caso de declarar as heranças de *B* e *C* como virtuais, assim nenhuma das duas teria uma "parte *A*":

```
class B: public virtual A {};
class C: public virtual A{};
class D: public B, public C {
 public:
   int valor() { return a; }
};
```

Agora a função *valor* não precisa mais do operador de escopo, e a árvore gerada terá apenas uma cópia de *A*.

Chamada de construtores de classes básicas virtuais

Como na herança virtual as classes derivadas não contém a parte da sua classe base, é preciso tomar alguns cuidados na hora de inicializar estas classes básicas. Supondo que a classe *A* tenha um construtor que receba um inteiro:

```
class A {
 public:
   int a;
   A(int) {}
};
```

A chamada a este construtor tem que estar explícita no código de *B* e *C*:

```
class B: public virtual A { public: B() : A(1) {} };
class C: public virtual A { public: C() : A(2) {} };
class D: public B, public C {
 public:
   int valor() { return a; }
   // erro! compilador não gera construtor vazio
};
```

Se um objeto da classe *B* for criado, sua parte *A* será inicializada com 1. Se for criado um do tipo *C*, a inicialização será com 2. E se o objeto for do tipo *D*? Como a parte *A* é criada diretamente por *D* (herança virtual), o próprio construtor de *D* deve chamar o de *A* diretamente:

```
class D: public B, public C {
public:
   int valor() { return a; }
   D(): A(3) {}
};
```

Exercício 10 - Classe List

Implementar uma lista que possa ser utilizada para armazenar qualquer objeto.

A lista só precisa ter um método para inserção de objetos. Além da classe lista, deve ser implementado um mecanismo para percorrer uma lista em dois sentidos: do início para o fim e do fim para o início. Isto não significa que a lista deve ser ordenada, apenas que ela preserva a ordem utilizada na inserção dos elementos. Uma maneira de preservar esta ordem é sempre inserir um elemento novo no início da lista.

Capítulo 7

Programação orientada a eventos

A programação tradicional

No passado, um sistema computacional era composto por uma CPU responsável por todo o processamento (mestre) e uma série de periféricos (escravos), responsáveis pela entrada e saída de dados.

Neste paradigma, o usuário assumia uma posição de periférico, já que ele era somente um mecanismo de entrada de dados.

O custo/benefício de uma CPU era extremamente superior ao de qualquer equipamento ou (custo de trabalho) usuário a ela conectados. Consequentemente, todos os sistemas davam prioridade máxima ao processamento, em relação à entrada e saída de dados.

Com isso, era comum encontrarmos sistemas em que a fatia de tempo dedicada aos periféricos era sensivelmente menor do que a dedicada à CPU.

Esta diferença de custos implicava uma menor dedicação, por parte dos programadores, aos periféricos e usuários em geral.

Ao colocar o usuário como escravo, este paradigma prejudicava a interação direta com o sistema.

O usuário não tinha como realizar de forma ágil suas operações, já que o sistema não priorizava a entrada e a saída de informações.

O diálogo dos usuários com o sistema foi evoluindo, ao longo dos tempos, porém continuava se apresentando como uma relação mestre-escravo, em que o primeiro ditava a seqüência da interação.

Mesmo os sistemas ditos avançados apresentavam uma hierarquia rígida de menus e/ou formulários, que eram preenchidos pelo usuário à medida que este navegava por essa estrutura.

Interfaces gráfico-interativas: o usuário como mestre

Com o avanço tecnológico, o custo/hora da CPU foi diminuindo, a ponto de se tornar cada vez mais comum encontrar sistemas onde o usuário determinava como se dava a interação e a navegação por suas estruturas.

Com o barateamento da tecnologia, os sistemas gráficos foram se mostrando uma forma mais razoável de representar estruturas complexas, onde a interação ocorre de forma **assíncrona**.

O conceito de eventos como paradigma de programação

O funcionamento de um sistema gráfico se dá através de ações sobre os diversos elementos da interface (objetos visuais). Cada uma destas ações corresponde um ou mais eventos “enviados” para esses elementos. Estes eventos devem ser identificados e tratados de maneira adequada pelo programa aplicativo.

Para manipular uma interface interativa, a programação tradicional se utiliza do recurso de um loop de controle principal (*case*/*switch*), que gerencia as opções de navegação oferecidas pelo sistema.

A necessidade de cuidar de eventos assíncronos com um paradigma síncrono acarreta uma relativa complexidade deste loop principal.

O conceito de objeto (versus procedimento) como mecanismo de programação

O conceito de objeto consiste basicamente em agrupar dados e código num mesmo elemento de programação. Ao invés de procedimentos atuando sobre estruturas de dados, passamos a ter estes dados (propriedades) associados a métodos (procedimentos). Estes métodos são os responsáveis pelo controle de fluxo e de informação entre os objetos do sistema.

Propriedades correspondem a campos de um registro (*record/struct/class*).

O conceito de **objeto visual** consiste na associação de uma representação gráfica (elemento de interface) a um trecho de código responsável pelo comportamento do objeto. A comunicação entre um objeto de interface e a aplicação se dá através da chamada de métodos de outros objetos (de interface ou não).

O pacote gráfico orientado a eventos DOSGRAPH

DOSGRAPH é uma biblioteca simples com apenas dez funções que possibilita uma programação orientada a eventos. A função principal é a de captura de eventos, e só retorna quando ocorre um evento. O eventos do DOSGRAPH são apenas os de mouse: click e unclick com os dois botões e mouse motion.

Uma aplicação típica que utiliza o DOSGRAPH (e qualquer outro pacote orientado a eventos) tem como núcleo um loop que captura os eventos e faz o processamento adequado. Todo o processamento é feito em resposta aos eventos.

Como aplicações que utilizam o mouse são tipicamente gráficas, o DOSGRAPH fornece várias funções de desenho que podem ser utilizadas, por exemplo dgLine, que desenha uma linha com a cor corrente. As funções do DOSGRAPH podem ser classificadas em:

- funções de desenho: dgLine, dgRectangle, dgFill, dgSetColor e dgSetMode;
- funções de consulta: dgWidth e dgHeight;
- funções de inicialização: dgOpen e dgClose;
- funções de controle: dgGetEvent.

Os tipos e funções definidos pelo DOSGRAPH estão definidos no apêndice A desta apostila.

Utilizando o DOSGRAPH em uma aplicação

Para fazer uma aplicação que use o DOSGRAPH, é preciso incluir a biblioteca no projeto. O nome da biblioteca é dg.lib, e deve ser incluída junto com os arquivos fonte que implementam a aplicação.

É importante lembrar que a função dgOpen deve ser chamada antes de qualquer outra função do DOSGRAPH. O resultado das funções antes de dgOpen é imprevisível.

Para executar a aplicação DOSGRAPH, é preciso que o arquivo egavga.lib esteja presente no diretório do executável. Caso este arquivo não esteja presente, a função dgOpen falha e gera uma mensagem de erro.

Exercício 11 - Uma aplicação orientada a eventos em C.

Implementar uma aplicação C que utilize o DOSGRAPH. Esta aplicação deve responder a um click com o primeiro botão desenhando um retângulo na posição do cursor do mouse. Um click com o segundo botão deve terminar a aplicação.

Exercício 12 - Classe Application

Implementar uma classe que represente uma aplicação DOSGRAPH. Sempre que for necessário fazer uma nova aplicação, basta criar uma nova classe derivada que responda aos eventos tratados, sem necessidade de usar as funções dgOpen, dgClose e dgGetEvent. Novas aplicações só precisam se preocupar com o processsamento dos eventos.

Capítulo 8

Projeto de uma aplicação orientada a objetos

Exercício 13 - Objetos gráficos que se movem na tela.

Implementar uma aplicação gráfica que permita drag de objetos. A organização do programa deve ser feita independentemente do tipo de objeto que será arrastado. Podem coexistir objetos diferentes na tela.

Basicamente o funcionamento deve ser:

- Assim que o programa começa, a tela está vazia;
- Um click com o botão esquerdo no fundo da tela cria objetos novos e
- Um click sobre um objeto começa o arraste.

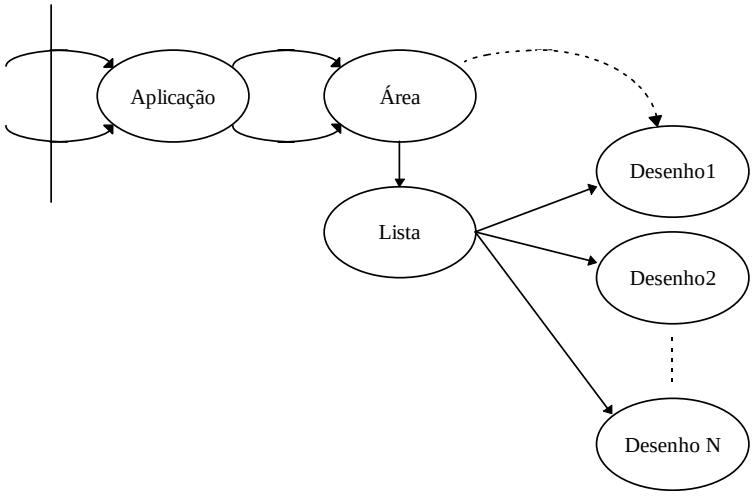
Atenção para a ordem dos objetos na tela: se dois estão sobrepostos, um click na área comum deve pegar o objeto visualmente acima. Para diferenciar a ordem dos objetos na tela, por exemplo use cores diferentes a cada criação.

Proposta para o exercício 13

A biblioteca dosgraph deve ser usada para a implementação desta aplicação.

Pode-se imaginar que um sistema destes é composto por uma área de trabalho, uma lista para guardar os objetos e os próprios objetos. Além disso, se uma aplicação dosgraph é representada por um objeto, este também é necessário.

A figura abaixo exemplifica uma possível implementação, com os objetos envolvidos e suas interações:



A estrutura apresentada esta no nível de projeto. Para a aplicação, é necessário definir a criação dos objetos e os próprios objetos em si, o que pode ser feito especializando-se algumas classes. Como a funcionalidade do drag independe do desenho, ela deve estar em alguma das classes acima. Coloque em *Desenho* todos os métodos necessários para esta funcionalidade, para que a *Area* possa manipulá-los.

Capítulo 9

Templates

Uma classe C++ normalmente é projetada para armazenar algum tipo de dado. Muitas vezes a funcionalidade de uma classe também faz sentido com outros tipos de dados. Este é o caso de muitos exemplos apresentados (por exemplo, *Pilha*).

Se uma classe é vista simplesmente como um manipulador de dados, pode fazer sentido separar a definição desta classe da definição dos tipos manipulados; isto é, pode-se fazer uma descrição de uma classe sem definir o tipo dos dados que ela manipula. Nesse caso, definição da classe é parametrizada por um tipo genérico T, sendo chamada C<T>. Esta construção não é uma classe realmente, mas uma descrição do conjunto de classes com o mesmo comportamento e que operam sobre um tipo T qualquer. Esta construção é denominada **class template**.

Com class templates, é permitido ao programador criar um série de classes distintas com a mesma descrição, mas sobre tipos distintos. Por exemplo, pode-se construir uma pilha de inteiros (Pilha<int>) ou pilha de strings (Pilha<char*>) apartir da mesma descrição. Esta descrição abstrata da template é utilizada pelo compilador para criar uma classe real em tempo de compilação, usando o tipo dos dados especificados quando de seu uso.

Em alguns textos acadêmicos, esta funcionalidade é chamada de polimorfismo paramétrico.

Declaração de templates

Consideremos novamente a classe *Pilha*, que já apareceu com várias implementações. Em todos os exemplos, a classe só armazenava inteiros, apesar de a funcionalidade nada ter a ver com o tipo de dado envolvido. Neste caso, ao invés de reescrevermos uma nova classe pilha para cada novo tipo demandado, pode-se definir uma template para a classe *Pilha*, onde o tipo armazenado é um T qualquer. A forma desta declaração é mostrada abaixo:

```
template<class T> class Pilha {
   struct  elemPilha {
      elemPilha* prox;
      T val;
      elemPilha(elemPilha*p, T v) { prox=p; val=v; }
   };
   elemPilha* topo;
 public:
   int  vazia()   { return topo == NULL }
   void push(T v) { topo = new elemPilha(topo, v); }
   T pop()
   { if (topo)
     {
        elemPilha *ep = topo;
        T v = ep->val;
        topo  = ep->prox;
        delete  ep;
        return v;
     }
     return -1;
   }
};
```

Tendo em vista que uma template é simplesmente uma descrição de uma classe, é necessário que toda esta descrição tenha sido lida antes de alguma declaração que envolva esta template. Isto significa, em se tratando de templates, que é necessário colocar no arquivo .h não apenas a declaração de classe, mas também a implementação de seus métodos.

Outra consequência de templates serem apenas descrições é que erros semânticos só aparecem na hora de usar a template. Durante a declaração da template apenas erros de sintaxe são checados. Mesmo que a template em si tenha sido compilada sem erros, podem aparecer erros quando de sua utilização.

Esta checagem semântica é realizada todas as vezes que a template é instanciada para algum tipo novo. Isto porque na definição do código da template não há nenhuma restrição quanto às operações que podem ser aplicadas ao tipo T. A checagem então tem que ser feita para cada tipo.

Usando templates

Definida a implementação de nossa template de pilhas, pode-se utilizá-la para qualquer tipo T criando pilhas de inteiros, strings,etc. Na criação de objetos destas classes pilhas, é necessário especificar o tipo de pilha na declaração do objeto:

```
void main()
{
   Pilha<int> intPilha;             // pilha de inteiros
   Pilha<char*> stringPilha;        // pilha de strings
   Pilha<Pilha<int>> intPilhaPilha; // pilha de pilha de inteiros

   intPilha.push(10);
   stringPilha.push( "teste" );

   intPilhaPilha.push( intPilha );
}
```

Declaração de métodos não inline

Quando da declaração de uma template, não é obrigatoria a implementação de seus métodos na forma inline; isto é, sua implementação pode não estar no corpo da declaração da classe. Para tal, basta especificar, de maneira análoga a descrição de classes, a template a que o método pertence.

Segue abaixo a implementação do método pop fora do corpo da template:

```
template<class T> class Pilha {
// estrutura interna da template...
 public:
   int  vazia()   { return topo == NULL }
   void push(T v) { topo = new elemPilha(topo, v); }
   T pop();       // protótipo do método pop
                  // implementado abaixo (fora da classe)
};
// Fim da declaração da classe
template<class T> T Pilha<T>::pop()
{ if (topo)
  { elemPilha *ep = topo;
    T v = ep->val;
    topo  = ep->prox;
    delete  ep;
    return v;
  }
  return -1;
}
```

Templates com vários argumentos genéricos

Templates não estão limitadas a terem apenas um único argumento. Pode-se utilizar diversos argumentos para parametrizar a descrição da classe. No exemplo abaixo, A *Pilha* armazena dois tipos de dados e ambos são parâmetros da template:

```
template<class T, class R> class Pilha {
   struct  elemPilha {
      elemPilha* prox;
      T t_val;
      R r_val;
      elemPilha(elemPilha*p, T t, R r)
         { prox=p; t_val=t; r_val=r; }
   };
   elemPilha* topo;
 public:
   int  vazia()     { return topo == NULL }
   void push(T t, R r) { topo = new elemPilha(topo, t, r); }
```

```
   void pop(T& t, R& r);
};
```

Templates com argumentos não genéricos

Templates não estão limitadas a argumentos genéricos; ou seja, tipos definidos pelo programador. Pode-se utilizar como argumentos da template tipos primitivos da linguagem como inteiros ou characteres. No exemplo abaixo, tem-se uma pilha que armazena um número fixo de elementos de um mesmo tipo:

```
template<class T, int S> class Pilha {
   struct  elemPilha {
      elemPilha* prox;
      T t_val[S];
      elemPilha(elemPilha*p, T* t);
   };
   elemPilha* topo;
 public:
   int  vazia();
   void push(T* t);
   void pop(T* t);
};
```

Templates de funções

Templates também podem ser usadas para definir funções. A mesma motivação para classes vale neste caso. Algumas vezes funções realizam operações sobre dados sem utilizar o conteúdo deles, ou seja, indepentemente de que tipo de dado seja.

Suponha que precisamos testar a magnitude de dois elementos quaisquer. A seguinte macro resolve este problema:

```
#define max(a,b) ((x>y) ? x : y)
```

Por muito tempo macros como esta foram usadas em programas C, mas isto tem os seus problemas. A macro funciona, mas impede que o compilador teste os tipos dos elementos envolvidos. A macro poderia ser utilizada para comparar um inteiro com um ponteiro, sem que sejam gerados erros.

Poderíamos usar uma função como esta:

```
int max(int a, int b)
{
   return a > b ? a : b;
}
```

Mas esta função só funciona para inteiros. Se o nosso programa só trabalha com escalares, poderíamos escrever uma função que trabalhe com double:

```
double max(double a, double b)
{
   return a > b ? a : b;
}
```

Nesse caso, o compilador se encarrega de converter os tipos char, int etc. para double, e a função funcionaria para todos estes casos.

Existem pelo menos duas limitações nesta versão. Uma delas diz respeito ao tipo dos parâmetros: apenas tipos que podem ser convertidos para double podem usar esta função. Isto significa que objetos e ponteiros não podem ser utilizados. A outra limitação se refere ao tipo de retorno: este é sempre double, independente do tipo passado. Suponha que a função display seja sobrecarregada para imprimir vários tipos de dados. Agora considere o código:

```
display(max('1', '9'));
```

Apesar de estarmos trabalhando com caracteres, a função display a ser chamada será a versão que trabalha com double, e o resultado será 57.00, que é o código ASCII do caractere '9'.

A opção de sobrecarregar max para vários tipos também não é boa, pois teríamos que reescrever o código, que seria idêntico, para todos os tipos que quiséssemos usar. Ainda assim o problema não estaria resolvido, pois novos tipos não poderiam ser usados sem que se criasse outra versão sobrecarregada de max.

Na verdade, qualquer tipo de dado que possua operações de comparação pode ser usado com max, e sempre da mesma maneira. Não é possível escrever uma única função que trate todos os tipos, mas o mecanismo de templates possibilita descrever, para o compilador, como estas funções podem ser implementadas:

```
template<class T> T max( T a, T b )
{
```

```
    return a > b ? a : b;
}
```

Esta função faz sentido para qualquer tipo T. Na realidade existirá uma implementação para cada tipo que for usado com esta função dentro do programa, mas estas funções serão geradas transparentemente pelo compilador. O uso de templates de funções não exige que se especifique explicitamente os tipos genéricos na chamada, como em templates de classes. Basta usar como se existisse uma função específica para o tipo envolvido:

```
void main()
{
   printf("%c", max('1', '9'));
}
```

Repare que, apesar de a função ser usada para comparar caracteres, em nenhum momento aparece a palavra char. O compilador sabe qual max precisa ser chamada pelo tipo dos parâmetros usados. Por causa disto, os tipos genéricos de templates de funções devem sempre aparecer nos parâmetros da função. Caso isto não aconteça, será sinalizado um erro de sintaxe na linha da declaração da template.

Assim como em templates de classes, templates de funções podem ter vários parâmetros genéricos.

Exercício 14 - Classe Vector para qualquer tipo.

Implementar com templates uma classe *Vector* com a mesma funcionalidade da desenvolvida na seção de sobrecarga de operadores, mas que possa ser usada com qualquer tipo.

Tratamento de Exceções

Quando do desenvolvimento de bibliotecas, é possível escrever código capaz de detectar erros de execução mas, em geral, não é possível fazer seu tratamento. Por outro lado, o usuário de uma biblioteca é capaz de fazer o correto tratamento de uma exceção mas não é capaz de detectá-la.

O conceito de exceção é introduzido para ajudar neste tipo de problema. A idéia fundamental é que uma função que detecte um problema e não seja capaz de resolvê-lo "acuse a exceção" esperando que quem a chamou seja capaz de realizar o tratamento adequado.

Funcionamento básico

O funcionamento dos tratadores de exceção é composto de diversas etapas:

- Definição das exceções que uma classe pode levantar;
- Definição de quando uma classe acusa uma exceção;
- Definição do(s) tratador(res) das exceções.

A definição da execeções que podem ser levantadas é feita na construção da classe. Neste momento define-se as condições de erro e os representa sob a forma de uma classe. Desta forma, cada condição de exceção é descrita por uma classe de exceção. Por exemplo, considere como representar e tratar o erro de indexação fora dos limites de um array dada pela classe *Vector*:

```
class Vector {
   int* p;
   int sz;
 public:
   class Range{ }; // classe de tratamento de exceção que
                   // representa acesso com índice inválido
   int& operator[]( int i );
};
```

A classe *Range*, a princípio sem dados internos, representa a condição de erro para acesso com índice não válido (menor que zero, maior que o espaço alocado, etc.)

A definição dos momentos da exceção são também definidos na construção da classe. A acusação das exceções é feita normalmente nos métodos da classe que encontrem alguma situação de erro. O levantamento de uma exceção é feita pelo comando throw que, conforme no nosso exemplo, é acionado quando o índice do array é inválido. Nestas situações, os objetos da classe *Range* são utilizados como exceções e são acusados da seguinte forma:

```
int& Vector::operator[]( int i )
{
  if ( i>=0 && i<sz ) return p[i];
  throw Range();  // Range() cria um objeto da classe Range
```

```
    // Quando indexamos o vetor com limites inválidos é
    // acusada a exceção correspondente
    // se a função que chamou operator[] souber tratá-la,
    // teremos o tratamento adequado.
  }
```

A definição dos tratadores aparecem, normalmente, nas funções que utilizam serviços das classes que levantam exceções. No caso de C++, estas funções podem selecionar quais as exceções que serão tratadas e trechos onde estas podem surgir.

No nosso exemplo, a função que precisa detectar a utilização de índices fora do limite indica seu interesse pelo tratamento colocando código correspondente na seguinte forma:

```
  void f( Vector& v )
  {
    //.. código qualquer sem tratamento
    try{
      //.. código qualquer com tratamento
      operação_qualquer( v );
    }
    catch( Vector::Range ) {
      // Aqui se encerra o código de tratamento da exceção
      // Range.
      // A função operação_qualquer apresentou a exceção
      // que está sendo tratada.
      // Esse código somente será executado se e somente se
      // a operação_qualquer fizer uso de indexação inválida.
    }
    //.. código qualquer sem tratamento
  }
```

A construção

```
  catch( /* nome da exceção */ ) { /* código */ }
```

é denominada manipulador de exceção (exception handler). Esta construção somente pode ser utilizada depois de um bloco prefixado com a palavra reservada try ou após outra construção *catch*. Os parênteses encerram a declaração dos tipos de objetos aonde o manipulador pode executar. Se a função *operação_qualquer* ou qualquer outra função chamada por *operação_qualquer* causar uma indexação inválida no array, será gerada uma exceção que será pega pelo manipulador e seu código executado.

Se um manipulador pegou uma determinada exceção, esta foi devidamente tratada e qualquer outro manipulador ainda existente se torna irrelevante. Em outras palavras, apenas o manipulador mais recentemente encontrado pelo controle da linguagem será executado. Por exemplo, dado que a função *f* pega uma exceção *Vector::Range*, uma função que chame *f* jamais pegará a exceção *Vector::Range*.

```
  int ff( Vector& v )
  {
    try{
      f(v);
    }
    catch( Vector::Range )
    { // este código jamais será executado ...
    }
  }
```

Naturalmente, um programa é capaz de tratar diversas exceções. Esses erros são mapeados com nomes distintos. Continuando o exemplo de *Vector*, trataremos mais um caso: criação de array com tamanho fora dos limites. Temos:

```
  class Vector {
    int* p;
    int sz;
    int max 512; // número máximo de elementos
   public:
    class Range{ }; //classe de tratamento de exceção
    class Size{ }; //classe de tratamento de exceção

    int& operator[]( int i );
    Vector( int sz )
    {
       if ( sz < 0 || max < sz ) throw Size();
```

```
      // continuação do construtor...
  }
};
```

O usuário da classe *Vector* pode discriminar a exceção pondo diversos manipuladores dentro do bloco precedido por *try*:

```
void f( Vector& v )
{
  try {
    qualquer_operação(v);
  }
  catch( Vector::Range ) {
  // código de tratamento de indexação inválida
  }
  catch( Vector::Size ) {
  // código de tratamento para criação de vetor muito grande
  }
// esse código é executado se não tiver ocorrido nenhuma
// exceção.
}
```

Nomeação de exceções

Uma exceção é tomada pelo manipulador não pelo seu tipo mas sim por um objeto. Havendo necessidade de transmitir alguma informação do levantamento da exceção para o manipulador, é necessário haver algum mecanismo de colocar tal informação neste objeto. Isto é feito colocando-se campos dentro da classe que representa a exceção e tomando seus valores nos manipuladores. Para tomarmos estes valores nos handles, é necessária a definição de um nome para o objeto criado na acusação.

No exemplo criado, é importante saber qual o valor que foi usado como índice na exceção *Vector::Range* (indexação fora dos limites):

```
class Vector{
// ...
public:
  class Range {
   public:
     int index;
     // criação do campo que diz o valor inválido
     Range( int i ) { index = i; }
     // a criação do objeto indica o valor inválido
  };
  int& operator[]( int i )
  // ...
};

int& Vector::operator[]( int i )
{
  if ( 0<=i && i<sz ) return p[i];
  throw Range(i);
  // acusa-se a exceção indicando
  // o valor índice invalido ao construir Range
}
```

Para examinar o índice incorreto, o manipulador deve dar um nome ao objeto da exceção:

```
void f( Vector& v )
{ /...
  try {
    qualquer_operação(v);
  }
  catch( Vector::Range r ) {
  // ‘r’ é o nome do objeto Range acusado no operador []
    printf( “índice errado: %d \n”, r.index );
    exit(0);
  }
  // ...
};
```

É interessante notar que no caso de templates, tem-se a opção de nomear a exceção de modo que cada classe instanciada pela template tenha sua própria classe de exceção:

```
template<class T> class Allocator {
  // ...
  class Exhausted {};  // classe do tratamento de exceção
  // ...
};

void f (Allocator<int>& ai, Allocator<double>& ad )
{
  try {
    // ...
  }
  catch (Allocator<int>::Exhausted) {
  // ...
  // tratamento de inteiros
  }
  catch (Allocator<double>::Exhausted) {
  // ...
  // tratamento de doubles
  }
}
```

Alternativamente, uma exceção pode ser comum a todas as classes instanciadas pela template:

```
class Allocator_Exhausted {};
template<class T> class Allocator {
  // ...
};

void f( Allocator<int>& ai, Allocator<double>& ad )
{
  try {
    // ...
  }
  catch( Allocator_Exausted ) {
  // ...
  // tratamento para ambos os tipos
  }
}
```

Exercício 15 - Template Array com tratamento de exceções

Introduzir tratamento de exceções na template Array.

Agrupamento de exceções

Normalmente as exceções podem ser categorizadas em famílias. Por exemplo, pode-se imaginar um erro matemático que inclua as exceções de overflow, underflow, divisão por zero, etc. A exceção de erro matemático (*MATHERR*) pode ser determinada pelo conjunto de erros que podem ser produzidos em uma biblioteca de funções numéricas.

Uma maneira de fazer *MATHERR* é determiná-la como um tipo de todos os possíveis erros numéricos:

```
enum MATHERR { Overflow, Underflow, ZeroDivide };
```

e na função que trata os erros:

```
void f( .... )
{
  try {
    // ...
  }
  catch( MATHERR m ) {
    switch( m ) {
    case Overflow:
      // ...
    case Underflow:
      // ...
    case ZeroDivide:
      // ...
```

```
    }
  }
}
```

De outra maneira, C++ usa a capacidade de herança e de funções virtuais para evitar este tipo de switch. É possível a utilização de herança para descrever coleções de exceções. Por exemplo:

```
class MATHERR {};
class Overflow: public MATHERR {} ;
class Underflow: public MATHERR {} ;
class ZeroDivide: public MATHERR {} ;
// ....
```

Para este caso, existem muitas ocasiões em que deseja-se fazer o tratamento de *MATHERR* sem saber precisamente de que tipo é o erro. Com a utilização de herança, é possível dizer:

```
try {
  // ...
}
catch( Overflow ) {
//  tratamento de overflow ou tudo derivado de overflow
}
catch ( MATHERR ) {
//  tratamento de qualquer outro erro numérico
}
```

A organização de exceções em hierarquias pode ser importante para a robustez do código de um programa. Consideremos o tratamento de todas as exceções de nossa biblioteca numérica sem o agrupamento destas. Neste caso, as funções que utilizam esta biblioteca teriam que exaustivamente determinar e tratar toda a lista de erros.

```
try {
// ...
}
catch (Overflow) { /* ..... */ }
catch (Underflow) { /* ..... */ }
catch (ZeroDivide) { /* ..... */ }
// e todas as outras exceções!!!
```

Isto não somente é tedioso mas dá margem ao esquecimento de alguma exceção. Além, uma determinada função que desejar fazer o tratamento de qualquer erro numérico (sem saber que tipo de erro) precisa ser constantemente atualizada quando do aparecimento de novas exceções. Por exemplo: o logaritmo de número menor ou igual a zero; o que implica também em recompilação destas funções.

Neste sentido é muito mais prático fazer:

```
try {
// ...
}
catch (MATHERR) { /* ..... */ }
// trata qualquer erro matemático.
```

que garante sempre o tratamento sem a necessidade de manutenção do código quando da introdução de novas exceções.

Exceções derivadas

A utilização de hierarquias de exceções naturalmente direciona os manipuladores que estão interessados somente em um subconjunto da informação carregada pelas exceções. Em outras palavras, uma exceção é normalmente tomada por um manipulador da classe básica ao invés de um da classe exata. A semântica para a tomada de um manipulador e nomeação de uma exceção é idêntica para a passagem de argumentos de funções vista anteriormente. Por exemplo:

```
class MATHERR {
  // ...
  virtual void debug_print() {};
};

class int_overflow : public MATHERR {
public:
  char op;
  int opr1, opr2;
  int_overflow( const char p, int a, int b )
  { op=p; opr1=a; opr2=b; }
  virtual void debug_print() // redefinição de debug_print
```

```
  { printf(“ operador:%c:( %d, %d )”, op, opr1, opr2 ); }
};

void f()
{
   try{
    g();
  }
  catch( MATHERR m ) { /* ... */ }
}
```

Quando um manipulador *MATHERR* é encontrado, *m* é um objeto *MATHERR* mesmo que a chamada de *g* tenha acusado um *int_overflow*. Isto implica que a informação extra encontrada em *int_overflow* está inacessível; isto é, se dentro do tratador chamarmos a função *debug_print*, não conseguiremos ver nada sobre o erro de overflow de inteiros. Isto é devido ao fato do compilador não fazer late-binding com o objeto.

No entanto, ponteiros e referências podem ser utilizados para evitar esta perda de informação. Para tal, pode-se escrever:

```
int add( int x, int y )
{
  if ( x>0 && y>0 && x>MAXINT - y
    || x<0 && y<0 && x<MININT + y )
    throw int_overflow( '+', x, y );
  return x + y;
}

void f()
{
  try {
    add( 1, 2 ); // ok
    add( MAXINT, 3 ); // causa exceção
  }
  catch( MATHERR& m ) {  // recebe no manipulador uma referência
    // ...
    m.debug_print();
    // chama o método de int_overflow!!!
  }
}
```

Re-throw

Dada uma função que capture uma exceção, não é incomum para um manipulador chegar a conclusão que nada pode ser feito a respeito do erro. Neste caso, a coisa típica a ser feita é a acusação da exceção novamente (re-throw), esperando que outro manipulador possa fazê-lo melhor. Por exemplo:

```
void h()
{
  try {
    // ...
  }
  catch( MATHERR ) {
    if ( posso_tratar() ) tratamento();
    else throw; // re-throw
  }
}
```

Um *re-throw* é indicado pelo comando *throw* sem argumentos. A exceção de relevantamento é a exceção original tomada e não somente a parte que era acessível como *MATHERR*. Em outras palavras, se um *int_overflow* foi acusado, a função que chamou *h* pode ainda tomar um *int_overflow* que *h* tomou como *MATHERR* e decidiu reacusar.

```
void k()
{
  try {
    h();
    // ...
  }
  catch( int_overflow ) {
```

```
    // ...
  }
}
```

A versão abaixo deste tipo de comportamento pode ser útil. Assim como em funções, pode-se utilizar '...' (indicando qualquer argumento) de modo que catch(...) signifique qualquer exceção. Por exemplo:

```
void m()
{
  try {
    // ...
  }
  catch(...) {
    limpeza();
    throw;
  }
}
```

Isto é, se qualquer exceção ocorrer, resultado da execução de parte de *m()*, a função *limpeza*() será chamada no manipulador e a exceção que causou a chamada da função *limpeza*() será reacusada.

Devido ao fato de que exceções derivadas podem ser tratadas por manipuladores para mais de um tipo de exceção, a ordem em que os estes aparecem após o bloco de try é relevante. Os tratadores são escolhidos em ordem. Por exemplo:

```
try {
  // ...
}
catch( ibuf ) {
  // tratador de input overflow
}
catch( io ) {
  // tratador de qualquer erro de I/O
}
catch( stdlib ) {
  // tratador de qualquer erro em bibliotecas
}
catch( ... ) {
  // tratador de qualquer outra exceção
}
```

Especificação de exceções

A acusação e o tratamento de exceções afetam o relacionamento entre as funções. Neste sentido, é interessante haver um mecanismo de especificar quais exceções que podem ser levantadas como parte da declaração de uma função. Por exemplo:

```
void f( int a ) throw (x2, x3, x4);
```

especifica que a função *f* só pode acusar as exceções *x2, x3, x4* e suas derivadas, nada mais. Deste modo, está garantindo para quem a chama que durante sua execução nenhuma outra exceção será levantada. Se, por acaso, algo acontecer que invalide esta garantia, a tentativa de acusação de uma exceção indevida será transformada em uma chamada para a função *unexpected*. O significado default para *unexpected* é a chamada a *terminate*, que normalmente representa um *abort*.

Desta forma, escrever:

```
void f( int a ) throw (x2, x3, x4)
{ /* implementacao qualquer */ }
```

significa a mesma coisa que:

```
void f( int a )
{
  try{
    /* implementação qualquer */
  }
  catch(x2) { throw; } // re-throw
  catch(x3) { throw; } // re-throw
  catch(x4) { throw; } // re-throw
  catch(...){ unexpected(); }
}
```

Mais que economia de digitação, o uso de especificação de exceções explicita as exceções que podem surgir na definição da função (.h) o que nem sempre aconteceria se esta definição ficasse em sua implementação.

Uma função sem especificação pode levantar qualquer exceção.

```
int f ();
```

enquanto que uma função sem a possibilidade de acusar qualquer exceção é declarada com uma lista explicitamente vazia:

```
int g() throw();
```

Exceções indesejadas

O mau uso de especificações de exceções pode levar a chamadas a função *unexpected*, que é indesejável a não ser no caso de testes. Pode-se evitar isto por uma boa estruturação e organização das exceções ou pela interceptação das chamadas a *unexpected*.

A função *set_unexpected* serve para interceptarmos estes casos. Esta função redefine o comportamento do sistema quando de uma exceção indesejada retornando o tratador antigo.

Abaixo temos um exemplo deste mecanismo. Neste caso, cria-se uma classe que representa um trecho aonde exceções não previstas devem ser tratadas.

```
typedef void(*functype)();
functype set_unexpected( functype );

class MyPart {
  functype old;
public:
  MyPart( functype f ) { old = set_unexpected( f ); }
  ~MyPart() { set_unexpected( old ); }
}

void new_trat() { printf("novo tratamento.\n"); }

void f()
{
  MyPart( &new_trat ); // construtor implica em redefinição
  g();
} // destrutor reseta unexpected() anterior
```

Neste caso, a execução de *f* é protegida contra erros de exceções não desejadas.

Exceções não tratadas

Uma exceção acusada e não tratada implica na chamada da função *terminate*. Esta também é chamada se o mecanismo de exceções de C++ encontrar a pilha corrompida. *terminate* executa a última função recebida como argumento da função *set_terminate*.

Este mecanismo serve como mais um nível para erros de exceção. É normalmente utilizado para medidas mais drásticas no sistema como: aborto da execução do processo, reinicialização do sistema, etc.

A redefinição deste comportamento é feita de modo análogo ao *unexpected*.

```
typedef void (*PFV) ();
PFV set_terminate (PFV);
```

Exemplo: calculadora RPN com tratamento de exceções

Como a calculadora usa a classe *Stack*, esta será atualizada para gerar exceções em caso de erro. Os erros possíveis são *push* com pilha cheia ou *pop* com pilha vazia:

```
// Excecoes da pilha
class StackError{};
class StackFull : public StackError {
 public:
   int size;
   StackFull( int s ) { size = s; }
};
class StackEmpty : public StackError {};

class Stack {
   friend class StackIterator;
   int size, top, *elems;
 public:
   Stack(int s = 50) { size = s; top = 0; elems = new int[size]; }
```

```
   ~Stack()         { delete elems; }
   void push(int i) throw (StackFull)
   { if (top>=size) throw StackFull(size); else elems[top++]=i; }
   int  pop() throw (StackEmpty)
   { if (top==0) throw StackEmpty(); else return elems[--top]; }
   int  empty()     { return top == 0; }
};
```

A própria calculadora tem seus erros:

```
class RpnError{};
class RpnNoOperands : public RpnError, public StackEmpty {};

class RPN : public Stack {
   void getop(int* n1, int* n2)  throw(RpnNoOperands);
 public:
   void sum() { int n1, n2; getop(&n1, &n2); push(n1+n2); }
   void sub() { int n1, n2; getop(&n1, &n2); push(n1-n2); }
   void mul() { int n1, n2; getop(&n1, &n2); push(n1*n2); }
   void div() { int n1, n2; getop(&n1, &n2); push(n1/n2); }
};

void RPN::getop(int* n1, int* n2) throw(RpnNoOperands)
{
   try {
      *n2 = pop();
      *n1 = pop();
   }
   catch ( StackEmpty )
   { throw RpnNoOperands(); }
}
```

Capítulo 10

Recursos novos de C++

Os recursos descritos nesta seção são relativamente novos e podem não estar implementados em todos os compiladores. No entanto, já foram adicionados ao padrão e merecem ser analisados.

Informação de tempo de execução

C++ permite que as classes tenham informação sobre o seu tipo dentro do objeto. Esta informação pode ser usada para testar em tempo de execução o tipo dos objetos, comparar se um objeto é subtipo de outro etc. Um dos problemas que este recurso resolve é a conversão de objetos de um tipo básico para um mais especializado. Com estas informações esta conversão passa a ser segura.

Como estas informações implicam em mais dados a serem armazenados nos objetos, normalmente o código é compilado sem estas informações. Como incluir estas informações no código gerado varia de compilador para compilador. O compilador Borland C++ (a partir da versão 4.0) possui uma opção de compilação para este controle. No Borland também é possível forçar uma classe a ter estas informações mesmo que a opção de compilação esteja desligada. A palavra reservada __rtti é usada na declaração das classes:

```
class __rtti A {
   // ...
};
```

As classes só conterão as informações se estas forem polimórficas, ou seja, se tiverem pelo menos um método virtual. Caso contrário, as informações não são geradas.

Consulta do tipo

O operador *typeid* é utilizado para obter o tipo de um objeto. O resultado de operador é um objeto da classe *Type_info*. Este objeto pode ser usado para comparação de tipos etc.:

```
#include <typeinfo.h>

class __rtti A {
   virtual void f() {}; // força A polimórfica
};

class B : public A {};

int main(void) {
   B b;
   B *bp = &b;

   if (typeid( *bp ) == typeid( B ) )
       printf("tipo de bp é %s\n", typeid( *Bptr).name());

   if (typeid( *bp ) != typeid( A ) )
       printf("bp não é do tipo A\n");

   return 0;
}
```

Este programa produz a saída:

```
tipo de bp é B
bp não é do tipo A
```

Type cast dinâmicos

A expressão

```
dynamic_cast<T>(ptr)
```

é utilizada para realizar conversões de tipos dinâmicas seguras. *T* deve ser um ponteiro ou uma referência para uma classe ou void*. *ptr* é uma expressão que gera um ponteiro ou uma referência.

Caso a conversão seja válida (o objeto apontado por *ptr* é realmente do tipo *T*), *ptr* é convertido para o tipo desejado. Se uma conversão de ponteiros for inválida, o resultado é 0. Se a conversão de uma referência falha, a exceção *Bad_cast* é sinalizada.

Conversões de um tipo derivado para um tipo básico são resolvidas em tempo de compilação. Conversões de uma classe básica para uma derivada, ou através das hierarquias são resolvidas em tempo de execução.

Exemplo:

```
#include <typeinfo.h>

class __rtti Base1
{
   virtual void f() {}; // força Base1 polimórfica
};

class __rtti Base2 { };
class Derived : public Base1, public Base2 { };

int main(void) {
 try {
   Derived d, *pd;
   Base1 *b1 = &d;

   // conversão da base para a derivada
   if ((pd = dynamic_cast<Derived *>(b1)) != 0)
      printf(“converteu para %s”, typeid(pd).name());
   else
      throw Bad_cast();

   Base2 *b2;
   // conversão através da hierarquia
   if ((b2 = dynamic_cast<Base2 *>(b1)) != 0)
      printf(“converteu para %s”, typeid(b2).name());
   else
      throw Bad_cast();
 }
 catch (Bad_cast) {
   printf("dynamic_cast não conseguiu converter\n”);
   return 1;
 }
 return 0;
}
```

Biblioteca de Streams

A biblioteca de entrada e saída padrão de C está obsoleta em C++. A linguagem oferece, como biblioteca padrão, um conjunto de classes para tratamento de entrada e saída. Entre as vantagens está a possibilidade de extensão para tratamento de tipos definidos pelo usuário e notação uniforme para todos os dispositivos.

O tratamento de tipos do usuário não era permitido em C pois as implementações são fixas. Não é possível usar a função printf para imprimir o valor de uma estrutura por exemplo. A notação uniforme também não acontecia, já que a implementação dos vários dispositivos pode ser totalmente diferente, o que acabava por gerar novas funções específicas para cada um.

Stream I/O

Streams trazem a elegância da sobrecarga de operadores para a parte de I/O. O objetivo das streams é uniformizar a notação dentre os diferentes tipos de I/O, deixando os detalhes para serem resolvidos pelo compilador. O uso de streams tem a seguinte notação:

```
stream_de_entrada >> variável;
stream_de_saída << variável;
```

O objeto stream é sempre colocado à esquerda na expressão. Os operadores << e >> são utilizados para indicar o fluxo de dados de um objeto para outro. Todos os tipos pré-definidos podem ser utilizados

com streams de I/O. Classes definidas pelo usuário também podem usar streams se as classes suportarem estas operações de I/O. O exemplo abaixo ilustra o uso de streams:

```
#include <iostream.h>

void main()
{
   int a;
   char c;
   float f;
   double d;

   cin >> a;
   cin >> c;
   cin >> f >> d;

   cout << a;
   cout << c << f << d;
   cout << "string";
}
```

C++ mantém a definição de entrada e saída de dados padrão, mas usa streams no lugar de arquivos. A stream cin é a entrada padrão, enquanto que cout é a saída padrão. Estas streams substituem stdin e stdout repectivamente, e são usadas com o mesmo propósito. Existem ainda duas streams cerr (semelhante a stderr) e clog, que não tem semelhante em ANSI C.

Assim como stdin, stdout e stderr, estas streams não precisam ser declaradas, inicializadas ou destruídas. Para usá-las, basta incluir o arquivo iostream.h.

Vamos agora analisar com mais calma o código apresentado. Streams não são mais do que uma biblioteca; ou seja, é um código normal como outro qualquer, que não precisa de nenhum tratamento especial por parte do compilador. Vamos entender como funciona uma stream.

O arquivo iostream.h define algumas classes, entre elas istream, ostream e iostream. istream é uma stream só de entrada, ostream só de saída e iostream é uma classe que herda de istream e ostream, servindo tanto para entrada como para saída. cin é um objeto de um tipo derivado de istream, que é declarado extern no arquivo iostream.h:

```
extern istream_withassign cin;
```

cin então é um objeto global, existindo durante todo o programa. Sendo global, o seu construtor é executado antes de main e o destrutor, depois de main. O construtor associa cin com a entrada padrão, e o destrutor desfaz esta associação. Daí não ser necessário se preocupar com a declaração, inicialização ou destruição.

cin funciona exclusivamente através de operadores. A classe istream define vários operadores >>, cada um recebendo um tipo diferente de parâmetro e retornando o próprio objeto. O que acontece no comando:

```
cin >> a;
```

é simplesmente a aplicação do método operator>> sobre o objeto cin passando como parâmetro a variável a. Já no seguinte caso:

```
cin >> a >> b;
```

o que acontece? É o mesmo caso de uma expressão

```
a + b + c;
```

Primeiro, a expressão a + b é avaliada; depois, o resultado é somado a c, resultando em (a + b) + c. Como o método operator>> retorna o próprio objeto (cin), a linha acima é o mesmo que

```
(cin >> a) >> b;
```

O mesmo acontece com cout.

I/O com classes definidas pelo usuário

As classes de streams permitem que classes definidas pelo usuário usem a mesma notação para qualquer tipo pré-definido. Basta sobrecarregar os operadores << e >>.

Vamos ver como estes operadores devem ser sobrecarregados. Supondo que a classe Ponto tenha os operadores, um uso típico seria:

```
int a, b;
Ponto p;
cin >> a >> p >> b;
```

Primeiro, será executado cin >> a. Ao resultado desta expressão (cin) será aplicado o operador >>, resultando em cin >> p. Fazendo esta análise, notamos que o operador não pode ser um membro na classe Ponto, já que o operador é aplicado a cin, e não a p. O operador deve ser definido assim:

```
istream& operator>> (istream& is, Ponto& p);
```

Eis a declaração completa de Ponto:

```
class Ponto {
   float x, y, z;
 public:
   Ponto(float a, float b, float c)
        { x=a; y=b; z=c; }
   friend ostream& operator<< (ostream& os, Ponto& p);
   friend istream& operator>> (istream& is, Ponto& p);
};

ostream& operator<< (ostream& os, Ponto& p)
{
   return os << '(' << p.x << ','
                    << p.y << ','
                    << p.z << ')';
}

istream& operator>> (istream& is, Ponto& p)
{
   return is >> p.x >> p.y >> p.z;
}
```

Além dos operadores << e >>, existem algumas funções. Por exemplo, a classe istream tem o métodos

```
istream& istream::putback(char);
istream& istream::getline(char*, int, char = '\n');
```

Manipuladores

A biblioteca ANSI C permite uma certa flexibilidade no modo como os dados podem ser formatados. Funções como printf, scanf etc. permitem especificar quantos digitos decimais devem ser mostrados, qual o tamanho do campo etc.

Com streams isto é feito através de manipuladores. Manipuladores são funções especialmente designadas para modificar o modo como uma stream trabalha. O arquivo iostream.h vem com uma série de manipuladores, e o arquivo iomanip.h define mais alguns.

Aqui está a lista dos manipuladores pré-definidos:

- dec: mostra os números na base decimal. Afeta int e long.
- hex: base hexadecimal
- oct: base octal
- ws: extrai brancos de uma istream
- endl: insere um caractere de fim de linha
- ends: insere um caractere de fim de string
- flush: decarrega os buffers de saída
- setbase(int): modifica a base. Aceita os valores 0 (default, =10), 8, 10 e 16.
- resetiosflags(long): limpa um ou mais flags de ios::x_flags
- setioflags(long): seta um ou mais flags de ios::x_flags
- setfill(int): define o caractere usado para preencher caracteres não usados quando a largura mínima é maior do que a utilizada; ver setw
- setprecision(int): define o número de dígitos decimais para float e double
- setw(int): define a largura da próxima variável a ser colocada em uma stream de saída.

Estes modificadores são usados assim:

```
cout << hex << 10 << setfill('.') << setwidth(10) << dec << 23;
```

A linha acima gera a seguinte saída:

```
a........23
```

Arquivos de entrada como streams

A elegância e a simplicidade das streams também pode ser utilizada para arquivos, tanto no modo texto como no binário. As classes utilizadas são ifstream e ofstream.

Arquivos são utilizados da mesma maneira que a apresentada nas seções anteriores, a única diferença é na hora da criação. Os construtores recebem o nome do arquivo a ser aberto:

```
#include <iostream.h>
#include <fstream.h>
```

```
void main()
{
   ifstream file("teste.c");

   if (!file)
   {
      return;
   }
   while (file)
   {
      char buffer[100];
      file.getline(buffer, 100);
      cout << endl << buffer;
   }
}
```

Existe um parâmetro default no construtor que não foi utilizado no exemplo acima, que serve para indicar que o arquivo é binário:

```
ifstream file(“teste.c”, ios::binary);
```

Testando erros em uma stream

Durante as operações, pode ocorrer uma situação de erro em uma stream. Pode-se tentar abrir um arquivo inexistente, ler depois do fim do arquivo etc. Todas estas condições causam erros, e devem ser detectadas e tratadas. A maneira mais simples de testar se ocorreu um erro é usando expressões da forma:

```
if (!file) // ocorreu um erro
```

ou

```
if (file) // nenhum erro
```

Expressões deste tipo são possíveis porque os operadores ! e void* são sobrecarregados em streams. Existem maneiras de investigar a causa do erro, como a função rdstate(), que retorna o tipo do erro. Outras funções estão disponíveis, como

```
if (file.bad()) // erro
if (file.eof()) // fim de arquivo
if (file.good()) // nenhum erro
```

Arquivos de saída como streams

O uso de arquivos de saída é semelhante aos de entrada. A diferença é que o parâmetro opcional do construtor pode receber outros valores além de ios::binary, e indicam o modo de abrir o arquivo:

```
// abre um arquivo para escrita,
// apaga se já existe
ofstream file1("teste.out");

// abre um arquivo para escrita no fim
ofstream file2("teste.out",ios::app);

// abre um arquivo vazio para escrita,
// se já existe gera erro
ofstream file3("teste.out",ios::noreplace);

// abre um arquivo vazio para escrita,
// erro se não existe
ofstream file4("teste.out",ios::nocreate);
```

Formatação na memória

Muitas vezes é preciso formatar dados sem escrever em um arquivo ou na saída padrão. Em ANSI C isto é feito com a função sprintf. A classe strstream pode ser usada para este fim. Ela funciona como qualquer stream com manipuladores etc. A classe istrstream tem a finalidade de ler ler uma string e extrair dados dela.

Exemplos de strstream:

```
#include <iostream.h>
#include <strstream.h>

void main()
{
   strstream buffer;

   int a = 20;
   float pi = 3.14159;

   buffer << "O número PI é " << pi << '.' << endl;
   buffer << "vinte (" << a << ")" << endl;

   cout << buffer.rdbuf();
}
```

Exemplos de istrstream:

```
#include <iostream.h>
#include <strstream.h>

void main(int argc, char *argv[])
{
   istrstream arg(argv[0]);

   cout << "Existem " << argc
        << "argumentos. O primeiro e "
        << arg.rdbuf();
}
```

Apêndice A - DOSGRAPH

Enumerações definidas pelo DOSGRAPH

DgMode

```
Valores: { dgCOPY, dgXOR }
```

DgColor

```
Valores: { dgBLACK,      dgBLUE,          dgGREEN,
           dgCIAN,       dgRED,           dgMAGENTA,
           dgBROWN,      dgLIGHT_GRAY,    dgGRAY,
           dgLIGHT_BLUE, dgLIGHT_GREEN,   dgLIGHT_CIAN,
           dgLIGHT_RED,  dgLIGHT_MAGENTA, dgYELLOW,
           dgWHITE }
```

DgEventType

```
Valores: { dgCLICK1, dgUNCLICK1, dgCLICK2, dgUNCLICK2, dgMOVE }
```

Tipo definido pelo DOSGRAPH

DgEvent

```
typedef struct {
   DgEventType tipo;
   int         x, y;
} DgEvent;
```

Funções definidas pelo DOSGRAPH

dgOpen

```
int  dgOpen(void);
```

Esta é a função de inicialização. Deve ser chamada antes de qualquer outra função do dosgraph.

dgClose

```
void dgClose(void);
```

Esta função finaliza o dosgraph. Após esta chamada, nenhuma função dosgraph deve ser chamada.

dgWidth

```
int  dgWidth(void);
```

Retorna a largura da tela em pixels.

dgHeight

```
int  dgHeight(void);
```

Retorna a altura da tela em pixels.

dgLine

```
void dgLine(int x0, int y0, int x1, int y1);
```

Desenha uma linha, com a cor e o modo corrente (ver dgSetColor e dgSetMode), do ponto (x0,y0) até (x1,y1).

dgRectangle

```
void dgRectangle(int x0, int y0, int x1, int y1);
```

Desenha um retangulo, com a cor e o modo corrente (ver dgSetColor e dgSetMode), do ponto (x0,y0) até (x1,y1).

O retângulo desenhado não é preenchido com a cor corrente, apenas as bordas são desenhadas.

dgFill

```
void dgFill(int x0, int y0, int x1, int y1);
```

Preenche uma área retangular na tela definida pelos pontos (x0,y0) e (x1,y1) com cor e modo corrente..

dgSetColor

```
void dgSetColor(DgColor cor);
```

Muda a cor corrente. Esta cor é utilizada pelas funções dgLine, dgRectangle e dgFill.

dgSetMode

```
void dgSetMode(DgMode modo);
```

Define o modo de desenho. Quando o modo é dgCOPY, a cor dos desenhos será a cor corrente. Caso o modo seja dgXOR, a cor do desenho será o resultado de uma operação xor entre a cor corrente e a cor do pixel na tela. Pela lógica do xor, se n1 xor n2 = n3, então n1 xor n3 = n2. O que significa que se um mesmo desenho for realizado duas vezes com a mesma cor e posição no modo xor, o desenho que estava na tela é restaurado. O primeiro desenho muda as cores, o segundo restaura. Por exemplo:

```
dgSetMode(dgXOR);
dgSetColor(dgWHITE);
dgRectangle(10,10,50,50);  // desenha um retangulo na tela.
                           // A cor resultante não será branca,
                           // e sim uma combinação de branco
                           // com o que já existia desenhado.
dgRectangle(10,10,50,50);  // restaura o desenho que existia na
                           // tela antes do primeiro GrRetangulo.
```

dgGetEvent

```
DgEventType dgGetEvent(DgEvent* ev);
```

Esta função captura um evento de mouse. Ao ser chamada, ela só retorna caso ocorra algum evento.

O tipo do evento é o retorno da função. No parâmetro ev são retornadas informações completas sobre o evento, no caso, a posição do cursor quando o evento foi gerado.

Os eventos podem ser:

- dgCLICK1 e dgCLICK2 quando um botão do mouse for pressionado;
- dgUNCLICK1 e dgUNCLICK2 quando um botão for solto e
- dgMOVE quando o cursor for movido.