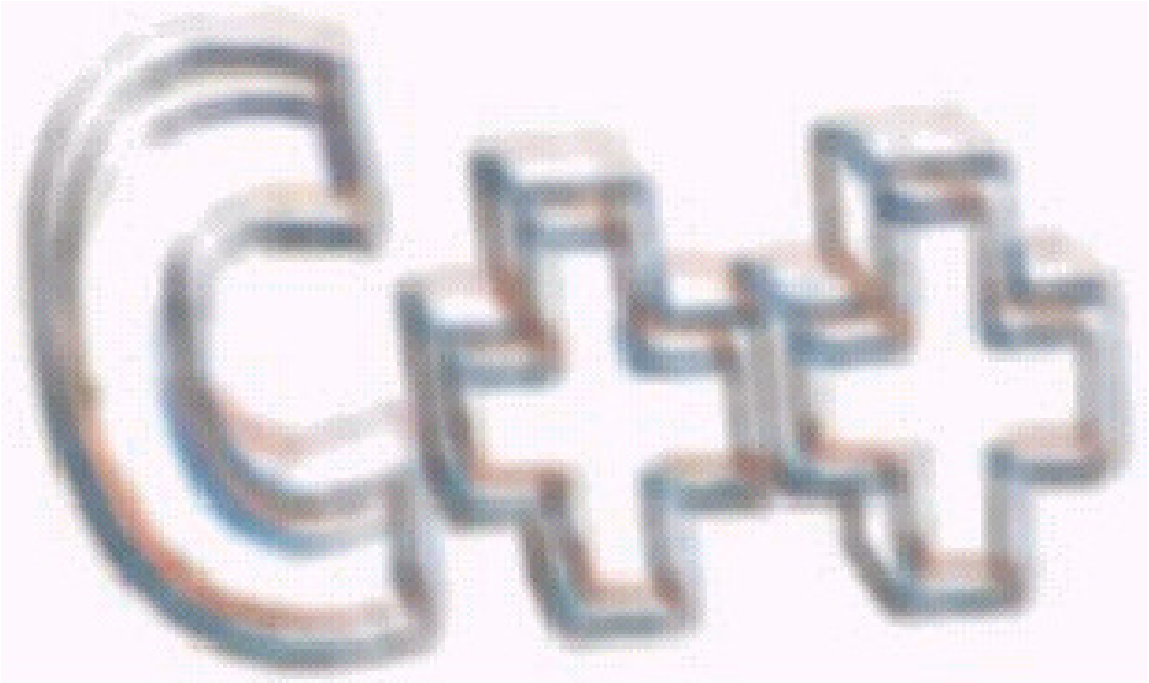
LINGUAGEM



Silvio do lago Pereira

Linguagem C++

Pereira, Silvio do Lago

Linguagem C++ /
Silvio do Lago Pereira. – São Paulo: FATEC, 1999.

1. Lógica de programação 2. Programação orientada a objetos 3. C++ (linguagem de programação para computadores) I. Título

Advertência

O autor acredita que todas as informações aqui apresentadas estão corretas e podem ser utilizadas para qualquer fim legal. Entretanto, não existe qualquer garantia, seja explícita ou implícita, de que o uso de tais informações conduzirá sempre ao resultado desejado.

Sumário

1. Um C melhorado

Esse capítulo apresenta os novos recursos oferecidos pela linguagem C++, sem no entanto explorar as características de orientação a objetos, que serão vistas brevemente.

1.1. Comentários

A linguagem C++ introduz uma nova forma de comentários. Além dos símbolos /* e */, oferecidos em C, podemos agora empregar também o símbolo //, a partir do qual todo o texto até o final da linha é ignorado.

Exemplo 1.1:

```
/* o comentário tradicional,
   englobando várias linhas,
   é válido tanto em C quanto em C++
*/

// já o comentário de uma única linha é válido apenas em C++
```

É preferível utilizar comentários no estilo C++ em vez daqueles no estilo C, que podem isolar importantes blocos de instruções.

1.2. Entrada e saída

A entrada e saída de dados em C é feita através das funções scanf() e printf(), disponíveis na sua biblioteca padrão. Apesar destas funções ainda estarem disponíveis em C++, vamos preferir utilizar o novo sistema de entrada e saída de dados por fluxo.

Uma vez incluído o arquivo iostream.h, um programa C++ dispõe de três fluxos predefinidos que são abertos automaticamente pelo sistema:

- cin, que corresponde à entrada padrão
- cout, que corresponde à saída padrão
- cerr, que corresponde à saída padrão de erros

O operador << permite inserir valores em um fluxo de saída, enquanto o operador >> permite extrair valores de um fluxo de entrada.

Exemplo 1.2:

```
#include <iostream.h>

void main(void)
{
   cout << "Olá mundo!\n";
}
```

O operador << pode ser usado, de forma encadeada, para inserir diversos valores em um mesmo fluxo de saída.

Exemplo 1.3:

```
#include <iostream.h>
void main(void)
{
   char nome[80];
   cout << "Qual o seu nome? ";
   cin >> nome;
   cout << "Olá " << nome << ", tudo bem?\n";
}
```

O operador >> também pode ser usado de forma encadeada.

Exemplo 1.4:

```
#include <iostream.h>
void main(void)
{
   float comprimento, largura;
   cout << "Informe o comprimento e a largura do retângulo: ";
   cin >> comprimento >> largura;
   cout << "Área do retângulo: " << comprimento * largura << " m2\n";
}
```

Note a ausência do operador de endereço (&) na sintaxe do fluxo cin. *Ao contrário da função* scanf(), *não precisamos usar o operador de endereço com o fluxo* cin.

Vantagens dos fluxos de entrada e saída:

- *execução mais rápida*: a função printf() analisa a cadeia de formatação durante a execução do programa, enquanto os fluxos são traduzidos durante a compilação;
- *verificação de tipos*: como a tradução é feita em tempo de compilação, valores inesperados, devido a erros de conversão, jamais são exibidos;
- *código mais compacto*: apenas o código necessário é gerado; com printf(), porém, o compilador deve gerar o código correspondente a todos os formatos de impressão;
- *uniformidade sintática*: os fluxos também podem ser utilizados com tipos definidos pelo usuário (através da sobrecarga dos operadores >> e <<).

Exemplo 1.5:

```
#include <stdio.h>
#include <iostream.h>
void main(void)
{
   int i = 1234;
   double d = 567.89;
   printf("\ni = %i, d = %d", i, d); // erro de conversão!
   cout << "\ni = " << i << ", d = " << d;
}
```

Observe a seguir como a função printf() exibe um valor inesperado para a variável d. Isso ocorre porque foi usado um formato de exibição errado (o correto seria %lf), que não pode ser verificado em tempo de execução.

```
i = 1234, d = -1202590843
i = 1234, d = 567.89
```

1.3. Manipuladores

Os manipuladores são elementos que determinam o formato em que os dados serão escritos ou lidos de um fluxo.

Os principais manipuladores são:

oct	leitura e escrita de um inteiro octal
dec	leitura e escrita de um inteiro decimal
hex	leitura e escrita de um inteiro hexadecimal
endl	insere um caracter de mudança de linha
setw(int n)	define campo com largura de *n* caracteres
setprecision(int n)	define total de dígitos na impressão de números reais
setfill(char c)	define o caracter usado no preenchimento de campos
flush	descarrega o *buffer* após a escrita

Exemplo 1.5:

```
#include <iostream.h>
#include <iomanip.h>
void main(void)
{
   int i=1234;
   float p=12.3456F;
   cout << "|"
        << setw(8) << setfill('*')
        << hex << i
        << "|\n"
        << "|"
        << setw(6) << setprecision(4)
        << p
        << "|" << endl;
}
```

Resultado da execução:

```
|*****4d2|
|*12.35|
```

1.4. Conversões explícitas

Em C++, a conversão explícita de tipos pode ser feita tanto através da notação de cast quanto da notação funcional.

Exemplo 1.6:

```
int i, j;
double d = 9.87;
i = (int)d;  // notação "cast"
j = int(d);  // notação funcional
```

Entretanto, a notação funcional só pode ser usada com tipos simples ou definidos pelo usuário. Para utilizá-la com ponteiros e vetores, precisamos antes criar novos tipos.

Exemplo 1.7:

```
typedef int * ptr;
int *i;
double d;
i = ptr(&d);  // notação funcional com ponteiro
```

1.5. Definição de variáveis

Em C++ uma variável pode ser declarada em qualquer parte do código, sendo que seu escopo inicia-se no ponto em que foi declarada e vai até o final do bloco que a contém.

Exemplo 1.8:

```
#include <iostream.h>
void main(void)
{
     cout << "Digite os valores (negativo finaliza): ";
     float soma = 0;
     while( true ) {
        float valor;
        cin >> valor;
        if( valor<0 ) break;
        soma += valor;
     }
     cout << "\nSoma: " << soma << endl;
}
```

Podemos até declarar um contador diretamente dentro de uma instrução for:

Exemplo 1.9:

```
#include <iostream.h>
void main(void)
{
     cout << "Contagem regressiva: " << endl;
     for(int i=9; i>=0; i--)
         cout << i << endl;
}
```

O operador de resolução de escopo (::) nos permite acessar uma variável global, mesmo que exista uma variável local com o mesmo nome.

Exemplo 1.10:

```
#include <iostream.h>
int n=10;
void main(void)
{
     int n=20;
     {
          int n=30;
          ::n++; // altera variável global
          cout << ::n << " " << n << endl;
     }
     cout << ::n << " " << n << endl;
}
```

A saída produzida por esse programa é a seguinte:

```
11 30
11 20
```

1.6. Constantes

Programadores em C estão habituados a empregar a diretiva #define do preprocessador para definir constantes. Entretanto, a experiência tem mostrado que o uso dessa diretiva é uma fonte de erros difíceis de se detectar.

Em C++, a utilização do preprocessador deve ser limitada apenas aos seguintes casos:

- inclusão de arquivos;
- compilação condicional.

Para definir constantes, em C++, usamos a palavra reservada const. Um objeto assim especificado não poderá ser modificado durante toda a sua existência e, portanto, é imprescindível inicializar uma constante no momento da sua declaração.

Exemplo 1.11:

```
const float pi = 3.14;
const int meses = 12;
const char *msg = "pressione enter...";
```

É possível usar a palavra const também na definição de ponteiros. Nesse caso, deve estar bem claro o que será constante: o objeto que aponta ou aquele que é apontado.

Exemplo 1.12:

```
const char * ptr1 = "um";          // o objeto apontado é constante
char * const ptr2 = "dois";        // o objeto que aponta é constante
const char * const ptr3 = "três"; // ambos são constantes
```

1.7. Tipos compostos

Assim como em C, em C++ podemos definir novos tipos de dados usando as palavras reservadas struct, enum e union. Mas, ao contrário do que ocorre na linguagem C, a utilização de typedef não é mais necessária para renomear o tipo.

Exemplo 1.12:

```
struct Ficha {
  char *nome;
  char *fone;
};
Ficha f, *pf;
```

Em C++, cada enumeração enum é um tipo particular, diferente de int, e só pode armazenar aqueles valores enumerados na sua definição.

Exemplo 1.13:

```
enum Logico { falso, verdade };
Logico ok;
ok = falso;
ok = 0;         // erro em C++, ok não é do tipo int
ok = Logico(0); // conversão explícita permitida
```

1.8. Referências

Além de ponteiros, a linguagem C++ oferece também as variáveis de referência. Esse novo recurso permite criar uma variável como sendo um sinônimo de uma outra. Assim, modificando-se uma delas a outra será também, automaticamente, atualizada.

Exemplo 1.14:

```
#include <iostream.h>
void main(void)
{
     int n=5;
     int &nr = n;     // nr é uma referência a n
     int *ptr = &nr;  // ptr aponta nr (e n também!)
     cout << "n = " << n << " nr = " << nr << endl;
     n += 2;
     cout << "n = " << n << " nr = " << nr << endl;
     *ptr = 3;
     cout << "n = " << n << " nr = " << nr << endl;
}
```

A saída produzida por esse programa é a seguinte:

```
n = 5 nr = 5
n = 7 nr = 7
n = 3 nr = 3
```

Uma variável de referência deve ser obrigatoriamente inicializada e o tipo do objeto referenciado deve ser o mesmo do objeto que referencia.

1.9. Alocação de memória

C++ oferece dois novos operadores, new e delete, em substituição respectivamente às funções malloc() e free(), embora estas funções continuem disponíveis.

O operador new aloca um espaço de memória, inicializa-o e retorna seu endereço. Caso a quantidade de memória solicitada não esteja disponível, o valor NULL é devolvido.

Exemplo 1.15:

```
int *p1      = new int;       // aloca espaço para um int
int *p2      = new int(5);    // aloca um int com valor inicial igual a 5
int *p3      = new int[5];    // aloca espaço para um vetor com 5 elementos
int (*p4)[3] = new int[2][3]; // aloca uma matriz de int 2x3
```

Cuidado para não confundir a notação:

```
int *p = new int(5);    // aloca espaço para um int e armazena nele o valor 5
int *q = new int[5];    // aloca espaço para um vetor com 5 elementos do tipo int
```

O operador delete libera um espaço de memória previamente alocado por new, para um objeto simples. Para liberar espaço alocado a um vetor, devemos usar o operador delete[]. A aplicação do operador delete a um ponteiro nulo é legal e não causa qualquer tipo de erro; na verdade, a operação é simplesmente ignorada.

Exemplo 1.16:

```
delete   p;    // libera um objeto
delete[] q;    // libera um vetor de objetos
```

É preciso observar que:

- a cada operação new deve corresponder uma operação delete;
- é importante liberar memória assim que ela não seja mais necessária;
- a memória alocada é liberada automaticamente no final da execução do programa.

2. Funções

Esse capítulo apresenta os detalhes envolvidos com a definição e uso de funções em C++; incluindo protótipos, passagem de parâmetros, funções inline e sobrecarga de funções.

2.1. Protótipos

Um *protótipo* é uma declaração que especifica a interface de uma função. Nessa declaração devem constar o tipo da função, seu nome e o tipo de cada um de seus parâmetros. Ao contrário da norma C-ANSI, que estabelece que uma função declarada com uma lista de argumentos vazia – f() – pode receber qualquer número de parâmetros, de quaisquer tipos, em C++, uma tal declaração equivale a f(void). Uma função cujo tipo não seja void deve, obrigatoriamente, devolver um valor através do comando return.

Exemplo 2.1:

```
int f(int,int);     // a função f recebe dois parâmetros int e retorna int
double g(char,int); // a função g recebe um char e um int e retorna double
```

A presença do nome de um parâmetro no protótipo é opcional; entretanto, recomenda-se que ele conste da declaração sempre que isso aumentar a legibilidade do programa.

Exemplo 2.2:

```
int pesquisa(char *lista[],int tam, char *nome);
void cursor(int coluna, int linha);
```

2.2. Passagem por referência

Além da passagem por valor, C++ também permite passar parâmetros por referência. Quando usamos passagem por referência, o parâmetro formal declarado na função é na verdade um sinônimo do parâmetro real que foi passado a ela. Assim, qualquer alteração realizada no parâmetro formal ocorre também no parâmetro real. Para indicar que um parâmetro esta sendo passado por referência, devemos usar o operador & prefixado ao seu nome.

Exemplo 2.3:

```
#include <iostream.h>
void troca(int &, int &);  // parâmetros serão passados por referência
void main(void)
{
   int x=5, y=7;
   cout << "Antes : x=" << x << ", y=" << y << endl;
   troca(x,y);
   cout << "Depois: x=" << x << ", y=" << y << endl;
}
```

```
void troca(int &a, int &b) // parâmetros recebidos por referência
{
   int x = a;
   a = b;
   b = x;
}
```

A execução do programa acima causa a seguinte saída:

```
Antes : x=5, y=7
Depois: x=7, y=5
```

A chamada de uma função com parâmetros de referência é muito simples. Essa facilidade aumenta a legibilidade e a potência da linguagem; mas deve ser usada com cautela, já que não há proteção ao valor do parâmetro real que é transmitido à função.

O uso da palavra reservada const na declaração dos parâmetros de referência permite anular o risco de alteração do parâmetro real, ao mesmo tempo em que evita que seu valor tenha que ser duplicado na memória, o que ocorreria numa passagem por valor.

Exemplo 2.4:

```
#include <iostream.h>
struct Ficha {
  char nome[20];
  char email[30];
};
void exibe(const Ficha &f)
{
   cout << f.nome << ": " << f.email << endl;
}
void main(void)
{
  Ficha usr = { "Silvio", "slago@ime.usp.br" };
  exibe(usr);
}
```

Referências e ponteiros podem ser combinados, sem nenhum problema:

Exemplo 2.4:

```
bool abrir(FILE *&arquivo, char *nome)
{
  if( (arquivo=fopen(nome,"r"))==NULL ) // se arquivo não existe
       arquivo=fopen(nome,"w");         // cria arquivo vazio
  return arquivo!=NULL;                 // informa se o arquivo foi aberto
}
```

Uma alternativa seria passar o parâmetro arquivo à função abrir() como um ponteiro de ponteiro, mas isso tornaria o código bem menos legível.

2.3. Parâmetros com valores *default*

Certos argumentos de uma função têm sempre os mesmos valores. Para não ter que especificar esses valores cada vez que a função é chamada, a linguagem C++ permite declará-los como *default*.

Exemplo 2.5:

```
#include <iostream.h>
#include <stdlib.h>
void exibe(int num, int base=10); // base tem valor default igual a 10
void main(void)
{
   exibe(13);      // decimal, por default
   exibe(13,2);    // binário
   exibe(13,16);   // hexadecimal
}
void exibe(int num, int base)
{
   char str[100];
   itoa(num,str,base);
   cout << str << endl;
}
```

Parâmetros com valores default devem necessariamente ser os últimos da lista e podem ser declarados tanto no protótipo quanto no cabeçalho da função, desde que tal declaração apareça antes de qualquer uso da função.

2.4. Funções *inline*

A palavra-chave inline substitui vantajosamente a utilização da diretiva #define do preprocessador para definir pseudo-funções, ou seja, macros.

Cada chamada de uma função inline é substituída por uma cópia do seu código, o que aumenta bastante a velocidade de execução do programa. Note porém que, por motivos de economia de memória, apenas funções muito curtas devem ser declaradas inline. A vantagem é que essas funções se comportam como funções normais e, portanto, permitem a consistência de tipo de seus parâmetros, o que não é possível com #define.

Exemplo 2.6:

```
#include <iostream.h>
inline double sqr(double n) { return n * n; }
void main(void)
{
   cout << sqr(10) << endl;
}
```

Ao contrário das funções normais, as funções inline somente são visíveis dentro do arquivo no qual são definidas.

2.5. Sobrecarga de funções

A assinatura de uma função se define por:

- o tipo de valor que ela devolve;
- seu nome;
- a lista de tipos dos parâmetros formais.

Entretanto, apenas os dois últimos desses atributos são discriminantes, isto é, servem para identificar que função está sendo chamada num ponto qualquer do código.

Podemos usar essa propriedade para dar um mesmo nome a duas ou mais funções diferentes, desde que elas tenham listas de parâmetros distintas. O compilador selecionará a função a ser chamada tomando como base o número e o tipo dos parâmetros reais especificados na sua chamada. Como essa escolha é feita em tempo de compilação, as funções sobrecarregadas têm desempenho idêntico às funções clássicas. Dizemos que as chamadas de funções sobrecarregadas são resolvidas de maneira estática.

Exemplo 2.7:

```
#include <iostream.h>
int soma(int a, int b)
{
   return a+b;
}
int soma(int a, int b, int c)
{
   return a+b+c;
}
double soma(double a, double b)
{
   return a+b;
}
void main(void)
{
   cout << soma(1,2) << endl;
   cout << soma(3,4,5) << endl;
   cout << soma(6.7,8.9) << endl;
}
```

3. Classes

Esse capítulo descreve as facilidades que C++ oferece para a criação de novos tipos de dados – classes – em que o acesso aos dados é restrito a um conjunto específico de funções de acesso.

3.1. Definição de uma classe

Uma *classe* descreve o modelo estrutural de um objeto, ou seja, define os *atributos* que o identificam e especifica que *métodos* podem operar sobre esses atributos.

Uma classe em C++ é uma estrutura que contém:

- dados membros, representando atributos;
- funções membros, representando métodos.

O exemplo a seguir mostra a declaração de uma classe que representa datas compactadas em 16 bits, segundo o esquema adotado pelo sistema DOS/Windows. Nesse esquema, o ano (menos 1980) é armazenado nos primeiros 7 bits da esquerda, o mês ocupa os próximos 4 bits e o dia ocupa o espaço restante. Note que a classe Data é definida como uma struct cujos membros são dados e funções.

Exemplo 3.1:

```
struct Data {
   unsigned short data; // |7|4|5|
   void define(short d, short m, short a);
   void exibe(void);
};
```

Ao contrário da linguagem C, além dos campos de dados, a linguagem C++ aceita também a declaração de funções dentro de uma estrutura.

O exemplo anterior mostra apenas a declaração da classe Data, sua implementação será vista mais adiante. Porém, supondo que podemos criar variáveis desse tipo, o próximo exemplo mostra como devemos chamar uma função membro.

Exemplo 3.2:

```
void main(void)
{
   Data hoje;
   hoje.define(5,1,2000);
   hoje.exibe();
}
```

A mesma notação usada para acesso aos dados deve ser usada para acesso às funções. Assim, se hoje é uma estrutura do tipo Data, então hoje.data permite acessar seu

campo de dados, enquanto hoje.define() e hoje.exibe() possibilitam chamar suas funções.

3.2. Encapsulamento

O *encapsulamento* consiste em impedir o acesso indevido a alguns atributos e métodos de uma classe. Em C++, os níveis de encapsulamento – *direitos de acesso* – são indicados através de três palavras-chaves:

- private: os membros privados somente são acessíveis às funções membros da classe. A parte privada é também denominada *implementação*.
- protected: os membros protegidos são como os membros privados, mas eles são também acessíveis às funções membros de classes derivadas.
- public: os membros públicos são acessíveis a todos. A parte pública é também denominada *interface*.

As palavras private, protected e public podem aparecer diversas vezes na declaração de uma classe. O direito de acesso permanece o mesmo até que um novo seja especificado.

Por *default*, todos os membros de uma struct são públicos. Assim, se quisermos evitar o acesso indevido ao atributo data, da classe definida no exemplo 3.1, podemos alterar sua declaração conforme a seguir:

Exemplo 3.3:

```
struct Data {
   void define(short d, short m, short a);
   void exibe(void);
   private:
      unsigned short data;
};
```

Agora, qualquer acesso ao campo data, fora das funções membros define() e exibe(), será impedido. Desta forma, mesmo sendo hoje uma variável da classe Data, a notação hoje.data causaria um erro de compilação.

Como um dos principais fundamentos da programação orientada a objetos é justamente o encapsulamento, C++ oferece uma nova palavra reservada – class – para a declaração de classes. A diferença é que numa classe definida com essa nova palavra, o acesso aos campos é privado por *default*, ao contrário do que ocorre com as estruturas.

Exemplo 3.4:

```
class Data {
   unsigned short data;
   public:
      void define(short d, short m, short a);
      void exibe(void);
};
```

O uso da palavra class tem como vantagem a garantia de que nenhum campo é deixado público por descuido. Entretanto, pode incentivar um estilo de codificação em que detalhes da implementação da classe aparecem antes da declaração de sua interface. Isso deve ser evitado, pois leva o usuário da classe a ter conhecimento de detalhes que

não são relevantes para quem a utiliza. É preferível listar primeiro os membros públicos e deixar os membros privados para depois.

Exemplo 3.5:

```
class Data {
   public:
      void define(short d, short m, short a);
      void exibe(void);
   private:
      unsigned short data;
};
```

3.3. Definição de funções membros

Em geral, a declaração de uma classe contém apenas os protótipos das suas funções membros. A definição dessas funções é feita num local distinto, no mesmo arquivo ou num arquivo separado. Para associar a definição de função membro à sua respectiva classe, devemos usar o operador de resolução de escopo (::). Dentro de uma função membro, temos acesso direto a todos os outros membros da classe.

Exemplo 3.6:

```
void Data::define(short d, short m, short a)
{
     data = ((a-1980)<<9) | (m<<5) | d;
}
void Data::exibe(void)
{
   cout << setw(2) << setfill('0')
        << (data & 0x1F) << "/"
        << setw(2) << setfill('0')
        << ((data>>5) & 0xF) << "/"
        << (data>>9)+1980 << endl;
}
```

A definição de uma função membro também pode ser feita diretamente dentro da declaração da classe. Nesse caso, uma tal função é automaticamente tratada pelo compilador como sendo inline. Uma função membro definida em separado também pode ser qualificada explicitamente como sendo inline.

Exemplo 3.7:

```
class Data {
   public:
      void define(short d, short m, short a)
      {
         data = ((a-1980)<<9) | (m<<5) | d;
      }
      void exibe(void);
   private:
      unsigned short data;
};
```

Lembre-se de que a visibilidade de uma função inline é restrita apenas ao módulo no qual ela é declarada, isso vale mesmo no caso de funções membros.

3.4. Instanciação de uma classe

De modo similar a uma estrutura, o nome de uma classe representa um novo tipo de dados. Podemos então definir variáveis desse novo tipo; tais variáveis são denominadas *instâncias* ou *objetos* da referida classe.

Exemplo 3.8:

```
Data hoje;                // uma instância simples (estática)
Data *ontem;              // um ponteiro, não inicializado
Data *amanha = new Data;  // criação dinâmica de uma instância
Data cronograma[31];      // um vetor de instâncias
```

Depois de criar um objeto, seja de maneira estática ou dinâmica, podemos acessar seus atributos e métodos. Esse acesso é feito como nas estruturas, através dos operadores ponto (.) ou seta (->).

Exemplo 3.9:

```
hoje.define(5,1,2000);    // ponto para instâncias
amanha->define(6,1,2000); // seta para ponteiros
cronograma[0].define(15,4,2000);
```

3.5. Funções membros constantes

Certos métodos de uma classe não devem modificar os dados da classe. Nesse caso, as funções que implementam tais funções devem ser declaradas como constantes. Esse tipo de declaração reforça o controle efetuado pelo compilador e permite uma programação mais segura, sem nenhum custo adicional de execução.

Exemplo 3.10:

```
void Data::exibe(void) const
{
   cout << setw(2) << setfill('0')
        << (data & 0x1F) << "/"
        << setw(2) << setfill('0')
        << ((data>>5) & 0xF) << "/"
        << (data>>9)+1980 << endl;
}
```

Apenas funções membros const podem ser chamadas a partir de objetos constantes. Entretanto, é possível sobrecarregar dois métodos com a mesma assinatura, desde que apenas um deles seja constante. Isso permite que o compilador selecione corretamente o método não constante para objetos não constantes, em vez de simplesmente gerar um erro de compilação.

3.7. Construtores e destrutores

Como os atributos de uma classe não podem ser inicializados diretamente na sua declaração, precisamos de um método para realizar essa tarefa, tal método é denominado *construtor*. De forma similar, alguns recursos precisam ser liberados quando um objeto não é mais necessário; um método que tenha essa finalidade é denominado *destrutor*.

O construtor é uma função membro que é chamada automaticamente sempre que uma nova instância de sua classe é criada, garantindo assim a correta inicialização dos objetos. Um construtor deve ter o mesmo nome da classe a qual pertence e não deve ter tipo de retorno, nem mesmo void.

Exemplo 3.11:

```
class Pilha {
   public:
      Pilha(); // construtor default
      ...
};
```

Um construtor que não tem parâmetros ou que tem valores *default* para todos eles é denominado *construtor default*. Se nenhum construtor é implementado pelo desenvolvedor da classe, então o compilador gera automaticamente um construtor *default*. Como as outras funções, os construtores também podem ser sobrecarregados.

Exemplo 3.12:

```
class Pilha {
   public:
      Pilha();      // construtor default
      Pilha(int n); // construtor com um parâmetro
      ...
};
```

O construtor é chamado apenas quando um objeto é instanciado, o que não acontece quando definimos um ponteiro para um determinado tipo de objetos.

Exemplo 3.13:

```
Pilha p;                                    // chama o construtor default
Pilha q(10);                                // chama construtor com um parâmetro
Pilha r = Pilha(10);                        // idem à instrução anterior
Pilha s[10];                                // chama construtor default 10 vezes
Pilha *t;                                   // não chama nenhum construtor
Pilha *u = new Pilha;                       // chama o construtor default
Pilha *v = new Pilha(5);                    // chama construtor com um parâmetro
Pilha x[3] = {Pilha(3), Pilha(9), Pilha(7)}; // inicia os elementos de x
```

Da mesma forma que o construtor, o destrutor é uma função membro que é chamada implicitamente sempre que um objeto de sua classe é destruído. Um destrutor deve ter como nome o nome de sua classe precedido por um til (~), não deve ter tipo de retorno nem parâmetros. Sendo assim, destrutores não podem ser sobrecarregados, já que não haveria forma do compilador decidir qual a versão a ser chamada.

Exemplo 3.14:

```
class Pilha {
   public:
      Pilha();   // construtor
      ~Pilha();  // destrutor
      ...
};
```

Como acontece no caso de construtores, o compilador também gera um destrutor *default*, caso o desenvolvedor da classe não especifique nenhum outro.

Construtores e destrutores são os únicos métodos não constantes que podem ser chamados a partir de objetos constantes.

A seguir temos uma implementação completa da classe Pilha.

Exemplo 3.15:

```
#include <iostream.h>
class Pilha {
   public:
      Pilha(int);
      ~Pilha(void);
      void insere(int);
      int  remove(void);
      int  topo(void);
      bool vazia(void);
      bool cheia(void);
   private:
      int  max;
      int  top;
      int *mem;
};
Pilha::Pilha(int n)
{
   max =  n;
   top = -1;
   mem = new int[n];
}
Pilha::~Pilha(void)
{
   delete[] mem;
}
void Pilha::insere(int e)
{
   if( !cheia() )
      mem[++top] = e;
   else
      cout << "pilha cheia!" << endl;
}
int Pilha::remove(void)
{
   if( !vazia() )
```

```
      return mem[top--];
   else {
      cout << "pilha vazia!" << endl;
      return 0;
   }
}
int Pilha::topo(void)
{
   if( !vazia() )
      return mem[top];
   else {
      cout << "pilha vazia!" << endl;
      return 0;
   }
}
bool Pilha::vazia(void)
{
   return top==-1;
}
bool Pilha::cheia(void)
{
   return top==max-1;
}
```

O exemplo a seguir mostra uma aplicação para a classe Pilha.

Exemplo 3.15:

```
// converte de decimal para binário
#include <iostream.h>
...
void main(void)
{
   Pilha p(32);
   unsigned  n;
   cout << "Número positivo? ";
   cin >> n;
   do {
      p.insere(n%2);
      n /= 2;
   } while( n!=0 );
   cout << "Binário: ";
   while( !p.vazia() )
      cout << p.remove();
   cout << endl;
}
```

3.9. Construtor de cópia

Considere a função exibeTopo(), definida no exemplo a seguir, que recebe uma pilha e exibe o elemento existente em seu topo.

Exemplo 3.16:

```
#include <iostream.h>
void exibeTopo(Pilha q)
{
  cout << "Topo da pilha: " << q.remove() << endl;
}
void main(void)
{
   Pilha p(5);
   p.insere(99);
   exibeTopo(p);
}
```

Como o passagem é por valor, o parâmetro formal q é criado como cópia temporária do parâmetro real p. O valor existente no topo da pilha é corretamente exibido mas, ao final da execução, o destrutor é chamado para liberar o espaço ocupado por q e acaba liberando também o espaço alocado para os dados membros de p. Isso acontece porque o objeto q contém ponteiros com os mesmos valores daqueles no objeto p.

Uma maneira imediata de resolver esse problema seria usar passagem por referência, o que impediria a chamada do destrutor ao término da execução da função; mas isso também causaria efeitos colaterais no valor do parâmetro real p, já que a função exibeTopo() remove um elemento da pilha q.

Uma solução efetiva requer, portanto, a criação de um *construtor de cópias*. Esse construtor é invocado quando criamos um objeto a partir de um outro objeto existente da mesma classe. O construtor de cópia é chamado também na passagem de argumentos por valor e no retorno de objetos.

Exemplo 3.17:

```
Pilha p(10);     // chama o construtor com um parâmetro
Pilha q(p);      // chama o construtor de cópia
Pilha r=p;       // idem ao caso anterior
```

Segue a codificação do construtor de cópia para classe Pilha:

Exemplo 3.18:

```
Pilha::Pilha(const Pilha &p)
{
   max = p.max;
   top = p.top;
   mem = new int[max];         // aloca novo espaço
   for(int i=0; i<=top; i++)  // copia os elementos
       mem[i] = p.mem[i];
}
```

3.10. Lista de inicialização de um construtor

A linguagem C++ diferencia inicialização de atribuição. A *atribuição* consiste em modificar o valor de uma variável, operação que pode ser efetuada diversas vezes. A *inicialização*, entretanto, é uma operação que é executada uma única vez, imediata-

mente após ter sido alocado espaço de memória para a variável. Essa operação consiste em dar um valor inicial a um objeto que está sendo criado.

Exemplo 3.19:

```
class A {
...
};
class B {
   public:
       B(int, int, int);
   private:
      const int x;
      int y;
      A z;
};
B::B(int a, int b, A c)
{
   x = a;  // erro: não é possível atribuir um valor a uma constante
   y = b;  // ok: atribuição legal
   ...     // como inicializar o objeto membro z ?
}
```

Para inicializar o dado membro constante `x` e chamar o construtor da classe `A`, a saída é usar uma lista de *inicialização*. Tal lista é especificada na definição do construtor e é usada na fase de inicialização de um objeto.

Exemplo 3.19:

```
B::B(int a, int b, A c) : x(a), y(b), z(c)
{
   // não há mais nada a fazer aqui!
}
```

A expressão `x(a)` indica ao compilador que o dado membro `x` deve ser inicializado com o valor do parâmetro `a` e a expressão `z(c)` indica que o dado membro `z` deve ser inicializado através de uma chamada ao construtor da classe `A` com o parâmetro `c`.

3.11. O ponteiro *this*

Todo método tem um parâmetro implícito denominado `this`. Esse parâmetro é um ponteiro contendo o endereço do objeto que chamou o método, permitindo assim que o método tenha acesso a todos os membros de tal objeto.

Exemplo 3.20:

```
int Pilha::tamanho(void)
{
    return this->max;   // o mesmo que escrever só max !
}
```

Uma função que retorne o ponteiro `this` pode ser usada de forma encadeada, uma vez que os operadores de seleção de membros são associativos da esquerda para a direita.

Exemplo 3.21:

```
Pilha *Pilha::ins(int x)
{
    if( !this->cheia() )
       this->mem[++this->top] = x;
    else
       cout << "Pilha cheia!" << endl;
    return this;
}
void main(void)
{
   Pilha *p = new Pilha(10);
   p->ins(11)->ins(22)->ins(33);  // chamadas encadeadas
   while( !p->vazia() )
      cout << p->remove() << endl;
}
```

3.12. Membros estáticos

Membros estáticos são úteis quando precisamos compartilhar uma variável entre os objetos de uma classe. Se declararmos um dado membro como estático, ele terá o mesmo valor para todas as instâncias da sua classe. A inicialização de tal dado deverá ser feita fora da classe, por uma declaração global.

Exemplo 3.22:

```
class Ex {
   public:
      Ex() { tot++; }
      ~Ex() { tot--; }
   private:
     static int tot;  // indica o total de instâncias existentes
};
int Ex::tot = 0;      // a inicialização deve ser feita assim!
```

Existe também a possibilidade de se declarar uma função membro estática. Uma tal função só tem acesso aos membros estáticos de sua classe, não pode ser sobrecarregada e pode ser chamada mesmo quando não existem instâncias de sua classe.

Exemplo 3.23:

```
#include <iostream.h>
class Ex {
   public:
      Ex() { tot++; }
      ~Ex() { tot--; }
      static void total()
      {
         cout << "Instâncias existentes: " << tot << endl;
      }
   private:
     static int tot;    // indica o total de instâncias existentes
};
int Ex::tot = 0;  // a inicialização deve ser feita assim!
```

```
void main(void)
{
    Ex::total();  // exibe 0, pois nenhuma instância foi criada
    Ex a, b, c;
    Ex::total();  // exibe 3
    a.total();    // idem ao caso anterior
}
```

3.13. Classes e funções amigas

Uma *função amiga* é aquela que, apesar de não ser membro da classe, tem livre acesso aos seus membros privados ou protegidos. Essa infração às regras de encapsulamento se justifica por razões de eficiência.

Exemplo 3.24:

```
class C {
   friend int manipula(C);
   public:
      ...
   private:
     int x;
}
int manipula(C c)
{
   c.x++;  // isso é permitido a uma função amiga!
}
```

É possível também declarar uma classe amiga de outra. Nesse caso, a classe amiga tem acesso livre a todos os membros privados da outra classe. No exemplo a seguir, a classe A pode acessar qualquer um dos membros da classe B.

Exemplo 3.25:

```
class A { ... }
class B {
   friend class A;
   ...
}
```

4. Sobrecarga

Esse capítulo descreve o mecanismo de sobrecarga, que permite redefinir o significado de um operador quando aplicado a objetos de uma determinada classe.

4.1. Sobrecarregando operadores

A sobrecarga de operadores possibilita uma sintaxe mais intuitiva quando trabalhamos com classes. Por exemplo, é muito mais claro e intuitivo adicionar dois números complexos m e n escrevendo r = m+n do que escrevendo adicionaComplexos(r,m,n).

Ao sobrecarregar um operador, tenha em mente que não é possível:

- alterar sua prioridade;
- alterar sua associatividade;
- alterar sua aridade;
- nem criar novos operadores.

Cada operador @ é representado por uma função denominada operator@. Por exemplo, quando o operador + é usado, o compilador gera uma chamada à função operator+. Um operador pode ser sobrecarregado tanto por uma função membro quanto por uma função amiga. Assim, uma expressão da forma a=b+c equivale a:

```
a = b.operator+(c);  // função membro
a = operator+(b,c);  // função amiga
```

A maioria dos operadores em C++ podem ser sobrecarregados, exceto os seguintes operadores: ::, ., ., ?: e sizeof. Além disso, os operadores =, (), [], ->, new e delete não podem ser sobrecarregados como funções amigas, só como funções membro.*

Para que a sobrecarga de operadores possa realmente tornar a leitura do código mais intuitiva, é importante nunca mudar a semântica do operador a ser sobrecarregado.

4.2. Sobrecarga por uma função membro

Como exemplo, vamos definir uma classe para representar números complexos e sobrecarregar o operador + para realizar a adição entre eles.

Exemplo 4.1:

```
class Complexo {
   public:
      Complexo(double a, double b) { r=a; i=b; }
      Complexo operator+(Complexo c) { return Complexo(r+c.r,i+c.i); }
      void exibe(void) { cout << r << (i<0 ? "-" : "+") << fabs(i) << "i\n"; }
   private:
      double r; // parte real
      double i; // parte imaginária
};
```

O programa a seguir exemplifica o uso da classe Complexo:

Exemplo 4.2:

```
#include <iostream.h>
#include <math.h>
...
void main(void)
{
   Complexo a(1,2);
   Complexo b(3,-4);
   Complexo c = a+b; // uso do operador sobrecarregado!
   a.exibe();
   b.exibe();
   c.exibe();
   (a+b+c).exibe();
}
```

Note que a expressão a+b, no exemplo acima, também poderia ter sido escrita como a.operator+(b). Quando o método operator+ é chamado, o seu parâmetro implícito this aponta o objeto a e o seu parâmetro formal c contém uma cópia do objeto b.

Sempre que for possível escolher, é preferível sobrecarregar um operador usando uma função membro, isso reforça o encapsulamento. Além disso, todos os operadores sobrecarregados por funções membros são transmitidos por herança (exceto atribuição).

É também possível sobrecarregar um operador mais que uma vez para uma determinada classe. Como exemplo, sobrecarregamos o operador + para que possamos adicionar números complexos a números reais.

Exemplo 4.3:

```
class Complexo {
   public:
      ...
      Complexo operator+(Complexo c) { return Complexo(r+c.r,i+c.i); }
      Complexo operator+(double d) { return Complexo(r+d,i); }
      ...
};
...
void main(void)
{
   Complexo a(1,2);
   Complexo b(3,4);
   (a+b).exibe();  // usa a primeira versão de operator+
   (a+5).exibe();  // usa a segunda versão de operator+
}
```

Na verdade, antes de chamar a segunda versão de operator+, o valor 5 é automaticamente promovido para o tipo double; então, é como se tivéssemos escrito a+5.0. Além disso, se a expressão a+5 fosse escrita como 5+a, teríamos um erro de compilação. Isso ocorre porque o operando à esquerda de um operador sobrecarregado por uma função membro deve ser sempre um objeto de sua classe.

4.3. Sobrecarga por uma função amiga

A sobrecarga por funções amigas é mais comum para operadores binários. De fato, ela permite aplicar as conversões implícitas ao primeiro membro da expressão.

Exemplo 4.4:

```
class Complexo {
   public:
      ...
      Complexo operator+(Complexo c) { return Complexo(r+c.r,i+c.i); }
      Complexo operator+(double d) { return Complexo(r+d,i); }
      friend Complexo operator+(double d, Complexo c);
      ...
};
Complexo operator+(double d, Complexo c)
{
         return Complexo(d+c.r, c.i);
}
...
void main(void)
{
   Complexo a(1,2);
   Complexo b(3,4);
   (a+b).exibe();  // usa a primeira versão de operator+
   (a+5).exibe();  // usa a segunda versão de operator+
   (7+b).exibe();  // usa a terceira versão de operator+
}
```

4.4. Operador de atribuição

O compilador constrói automaticamente, para cada classe, um operador de *atribuição default*. Entretanto, esse operador apenas duplica os valores dos dados membros do objeto sendo atribuído e, no caso de campos que guardam endereços, a área apontada não é corretamente duplicada. Novamente, temos o mesmo problema já visto quando discutimos o construtor de cópia para uma classe. Claramente, isso causará problemas quando um dos objetos envolvidos nesse tipo de atribuição for destruído.

O operador de atribuição deve ser obrigatoriamente uma função membro e, sendo `x`, `y` e `z` três objetos da mesma classe, deve funcionar corretamente nos dois casos seguintes:

- `x = x;`
- `x = y = z;`

Como a classe `Complexo` não usa alocação dinâmica, e portanto o operador de atribuição *default* deve funcionar perfeitamente bem, vamos retomar a classe `Pilhas`:

Exemplo 4.5:

```
class Pilha {
   public:
      Pilha operator=(const Pilha &);
      ...
};
```

```
Pilha Pilha::operator=(const Pilha &p)
{
   if( this != &p ) { // trata o caso x=x
      this->max = p.max;
      this->top = p.top;
      delete[] this->mem;
      this->mem = new int[max];
      for(int i=0; i<=top; i++)
         this->mem[i] = p.mem[i];
   }
   return *this; // trata o caso x=y=z
}
```

4.5. Sobrecarga de ++ e --

Os operadores de incremento e decremento são únicos em C/C++, já que apenas eles podem ser usados tanto de forma prefixada quanto posfixada.

Temos a seguir a sintaxe necessária para sobrecarregar o operador de incremento para uma determinada classe C, a sintaxe para o operador de decremento é análoga:

- *notação prefixada*:
  função membro: `C operator++();`
  função amiga: `C operator++(C &);`
- *notação posfixada*:
  função membro: `C operator++(int);`
  função amiga: `C operator++(C &,int);`

O argumento int é necessário para indicar que a função deverá ser invocada quando o operador é usado na forma posfixa. Esse argumento nunca é usado; ele serve apenas para distinguir a versão prefixada da posfixada.

Exemplo 4.6:

```
class Complexo {
   public:
      Complexo operator++(void) { this->r++; return *this; }
      Complexo operator++(int)  { this->r++; return *this; }
      ...
};

void main(void)
{
   Complexo a(1,2);
   Complexo b(3,4);
   Complexo c = ++a + b;  // notação prefixa
   Complexo d = a + b++;  // notação posfixa
   a.exibe();
   b.exibe();
   c.exibe();
   d.exibe();
}
```

4.6. Operadores de conversão

Dentro da definição completa de uma classe, não podemos esquecer de definir os operadores de conversão de tipos. Existem dois tipos de conversão:

- tipo predefinido em um tipo classe: feita pelo *construtor de conversão*.
- tipo classe em um tipo predefinido: feita por *funções de conversão*.

Um construtor com apenas um argumento do tipo T permite realizar uma conversão de uma variável do tipo T em um objeto da classe a que pertence o construtor.

Exemplo 4.7:

```
class Complexo {
   public:
      ...
      Complexo(double d) { r=d; i=0; }
      ...
};
void f(Complexo c) { ... }
void main(void)
{
   Complexo a = 3; // idem a(3.0,0);
   a.exibe();
   f(5);           // chama o construtor para converter 5 em 5+0i
}
```

Nesse exemplo, o construtor efetua conversões de double para Complexo, mas também pode ser usado para converter int em Complexo. Pelas regras de conversão de C++, um int é convertido automaticamente para double e, depois, o construtor termina o serviço.

Uma função de conversão é um método que efetua uma conversão de um tipo classe para um tipo primitivo T. Ela é nomeada operator T() e não pode ter tipo de retorno, embora devolva um valor do tipo T.

Exemplo 4.8:

```
class Complexo {
   public:
      ...
      operator double() const { return r; }
      ...
};
void main(void)
{
   Complexo a = 2;     // converte double para Complexo
   double d = 3*a;     // converte Complexo para double
   cout << d << endl;
}
```

5. Padrões

Esse capítulo introduz o conceito de padrões que permite criar, de maneira simples e eficiente, classes para armazenamento de coleções de tipo genérico. Similarmente, padrões permitem criar funções genéricas, que funcionam para uma família de tipos.

5.1. Padrões de funções

Um *padrão de função* é um mecanismo para gerar funções, usando tipos de dados como parâmetros. Por exemplo, para criar uma função que retorna o mínimo entre dois valores, sem usar um padrão, precisamos criar um conjunto de funções sobrecarregadas:

Exemplo 5.1:

```
char   min(char a, char b)     { return a<b ? a : b; }
int    min(int a, int b)       { return a<b ? a : b; }
long   min(long a, long b)     { return a<b ? a : b; }
float  min(float a, float b)   { return a<b ? a : b; }
double min(double a, double b) { return a<b ? a : b; }
...
```

Usando um padrão, entretanto, podemos eliminar essa redundância de código:

Exemplo 5.2:

```
template <class Tipo>
Tipo min(Tipo a, Tipo b)
{
   return a<b ? a : b;
}
```

O prefixo template <class T> especifica que um padrão está sendo criado e que um argumento T, representando um tipo de dados, será usado na sua definição. Depois dessa introdução, T pode usado da mesma forma que qualquer outro tipo de dados. O escopo do tipo T se estende até o final da definição prefixada com template <class T>.

Da mesma forma que funções normais, funções genéricas também podem ser sobrecarregadas. Isso nos permite especializar uma função para uma situação particular, como no exemplo a seguir:

Exemplo 5.3:

```
#include <iostream.h>
#include <string.h>
template <class Tipo> Tipo min(Tipo a, Tipo b) { return a<b ? a : b; }
char *min(char *a, char *b) { return strcmp(a, b)<0 ? a : b; }
void main(void)
{
   cout << min(35,28) << " " << min("alfa","beta") << endl;
}
```

Um padrão é um modelo a partir do qual o compilador poderá gerar as funções que forem necessárias. Quando um padrão é chamado pela primeira vez para um determinado tipo T, uma nova instância do padrão é criada pelo compilador. A partir daí, essa versão especializada é usada sempre que a função for chamada com um argumento do tipo T.

Exemplo 5.4:

```
#include <iostream.h>
template <class T>
void exibe(T *vet, int tam)
{
   for(int i=0; i<tam; i++)
      cout << vet[i] << " ";
  cout << endl;
}
void main(void)
{
  int x[5] = {17, 14, 65, 10, 71};
  exibe(x,5);
  double y[3] = {19.2, 91.6, 45.7};
  exibe(y, 3);
  char *z[] = {"Bjarne", "Stroustrup"};
  exibe(z, 2);
}
```

No exemplo acima, o compilador deverá criar três instâncias distintas da função exibe(): a primeira para o tipo int, a segunda para double e a terceira para o tipo char *.

Mesmo que várias instâncias idênticas sejam criadas, em arquivos separados, apenas uma dessas cópias fará parte do código executável gerado.

5.2. Padrões de funções *versus* macros

Um padrão funciona mais ou menos como uma macro; entretanto, há muita diferença entre uma macro como essa:

```
#define min(a, b) (((a)<(b)) ? (a) : (b))
```

e o padrão:

```
template <class Tipo> Tipo min(Tipo a, Tipo b) { return a<b ? a : b; }
```

Ao contrário do padrão, a macro apresenta os seguintes problemas:

- o compilador não tem como verificar se os parâmetros da macro são de tipos compatíveis, uma macro é expandida sem que nenhuma checagem de tipos seja feita;
- os parâmetros a e b são avaliados duas vezes; por exemplo, se o parâmetro é uma variável pos-incrementada, o incremento é feito duas vezes;
- como a macro é expandida pelo preprocessador, se ela apresentar algum erro de sintaxe, a mensagem emitida pelo compilador irá se referir ao local da expansão e não ao local onde a macro foi definida.

5.3. Classes parametrizadas

Padrões são uma excelente maneira de implementar uma coleção de classes. Eles nos permitem definir classes genéricas, ou *parametrizadas*, e depois passar a elas um tipo de dados como parâmetro, para que seja construída uma classe específica. Como exemplo, vamos criar uma classe parametrizada para pilhas:

Exemplo 5.5:

```
template <class Tipo>
class Pilha {
   public:
      Pilha(int);
      ~Pilha(void);
      void insere(Tipo);
      Tipo remove(void);
      Tipo topo(void);
      bool vazia(void);
      bool cheia(void);
   private:
      int  max;
      int  top;
      Tipo *mem;
};
```

Ao definir uma função membro de uma classe parametrizada, não devemos esquecer de que a definição deve ter template <class Tipo> como prefixo e que o nome da classe a que ela pertence deve ter <Tipo> como sufixo. Além disso, apesar de construtores e destrutores referenciarem o nome da classe duas vezes, o sufixo só aparecer na primeira.

Exemplo 5.6:

```
template <class Tipo> Pilha<Tipo>::Pilha(int n)
{
   top = -1;
   mem = new Tipo[max = n];
}
template <class Tipo> Pilha<Tipo>::~Pilha(void)
{
   delete[] mem;
}
template <class Tipo> void Pilha<Tipo>::insere(Tipo e)
{
   if( !cheia() )
      mem[++top] = e;
   else
      cout << "pilha cheia!" << endl;
}
template <class Tipo> Tipo Pilha<Tipo>::remove(void)
{
   if( !vazia() )
      return mem[top--];
   else {
      cout << "pilha vazia!" << endl;
      return 0;
   }
}
```

```
template <class Tipo> Tipo Pilha<Tipo>::topo(void)
{
   if( !vazia() )
      return mem[top];
   else {
      cout << "pilha vazia!" << endl;
      return 0;
   }
}
template <class Tipo> bool Pilha<Tipo>::vazia(void)
{
   return top==-1;
}
template <class Tipo> bool Pilha<Tipo>::cheia(void)
{
   return top==max-1;
}
```

5.4. Instanciação de classes parametrizadas

Ao contrário da instanciação funções, a instanciação de uma classe parametrizada exige que seja fornecido, explicitamente, um tipo de dados como argumento. Nenhum código é gerado para um padrão de classes até que seja feita uma instanciação.

Exemplo 5.7:

```
#include <iostream.h>
...
void main(void)
{
   Pilha<char> p(5); // cria uma pilha de char
   p.insere('a');
   p.insere('b');
   p.insere('c');
   while( !p.vazia() )
      cout << p.remove() << endl;
   Pilha<double> q(5); // cria uma pilha de double
   q.insere(1.2);
   q.insere(3.4);
   q.insere(5.6);
   while( !q.vazia() )
      cout << q.remove() << endl;
}
```

Funções membros de classes parametrizadas somente são instanciadas quando são chamadas; isso pode causar problemas se você tentar criar uma biblioteca, usando padrões, para outros usuários. No exemplo 5.7, o código para as funções `topo()` *e* `cheia()` *não seria gerado; já que tais funções não foram chamadas.*

6. Herança

Esse capítulo apresenta as classes derivadas, que proporcionam um mecanismo simples, flexível e eficiente para a criação de novas classes a partir de outras classes já existentes, sem a necessidade de alteração ou recompilação de código.

6.1. Herança simples

Herança é um mecanismo que nos permite criar uma nova classe a partir de uma outra classe já existente, a *classe base*. A nova classe, ou *classe derivada*, herda automaticamente todos os membros da sua classe base.

Por exemplo, podemos definir uma classe Pixel herdando características da classe Ponto. Essa idéia parece bastante natural, se consideramos que um *pixel* é apenas um ponto que tem uma cor e que pode estar aceso ou apagado.

Exemplo 6.1:

```
class Ponto {
   public:
      Ponto(int x, int y);
   private:
      int x;
      int y;
};
```

Para indicar que a classe Pixel é derivada da classe Ponto, escrevemos Pixel : Ponto.

Exemplo 6.2:

```
class Pixel : Ponto {   // Pixel é derivada de Ponto!
   public:
      Pixel(int x, int y, int c);
      void acende(void);
      void apaga(void);
   private:
      int cor;
};
```

Geralmente, é mais fácil adaptar que reinventar. Graças à herança, ao definirmos uma classe derivada, precisamos apenas adicionar os novos membros desejados e, às vezes, redefinir alguns métodos já existentes para que funcionem conforme esperado.

Cada classe derivada é uma classe completamente independente e nenhuma alteração feita por ela pode afetar sua classe base. Além disso, o conceito de classe base e derivada é relativo, uma classe derivada num contexto pode servir de classe base em outro.

6.2. Modos de derivação

Quando criamos uma classe derivada, podemos especificar um *modo de derivação* usando uma das seguintes palavras reservadas: public, protected ou private. Esse modo determina a acessibilidade dos membros herdados dentro da classe derivada. Se nenhum modo é especificado, o compilador C++ assume o modo private para classes e public para estruturas. Os membros privados da classe base jamais são acessíveis aos membros da classe derivada, não importando o modo de derivação indicado.

Segue uma breve descrição dos modos de derivação:

- *Herança pública*: os membros públicos e protegidos da classe base têm a mesma acessibilidade dentro da classe derivada. Esse é o tipo mais comum de herança, pois permite modelar relações da forma "*Y é um tipo de X*" ou "*Y é uma especialização da classe X*".
- *Herança protegida*: membros públicos e protegidos na classe base tornam-se membros protegidos na classe derivada. Os membros herdados são acessíveis na implementação da classe base, mas não faz parte de sua interface.
- *Herança privada*: membros públicos e protegidos na classe base tornam-se privados na classe derivada. Esse tipo de herança permite modelar relações da forma "*Y é composto de X*" e, portanto, equivale a fazer da classe base um dado membro da classe derivada, ou seja, podemos usar *composição* em vez de herança privada.

Para especificar um modo de derivação, devemos empregar a seguinte sintaxe:

```
class <derivada> : <modo> <base> {
   ...
};
```

A tabela a seguir exibe um resumo da acessibilidade dos membros herdados por uma classe derivada, em função do modo de derivação escolhido:

		Status na classe base	*Status na classe derivada*
Modo de derivação	public	public	public
		protected	protected
		private	---
	protected	public	protected
		protected	protected
		private	---
	private	public	private
		protected	private
		private	---

6.3. Redefinição de métodos numa classe derivada

É perfeitamente possível redefinir numa classe derivada um método que tenha sido herdado de sua classe base, bastando para isso criar um outro método com o mesmo nome. Nesse caso, se for necessário usar na classe derivada o método que foi herdado, será preciso usar o operador de resolução de escopo (::). Por exemplo, suponha que a classe base `B` tenha um método `m()` e que esse método tenha sido redefinido na sua classe derivada. Então, para acessar o método da classe base dentro da classe derivada, basta escrever `B::m()`.

Para exemplificar a redefinição de um método numa classe derivada, vamos primeiro criar a classe base, que representará um tipo de vetor que encapsula o seu tamanho como parte de sua estrutura.

Exemplo 6.3:

```
class Vetor {
   public:
      Vetor(int n);
      Vetor(const Vetor &);
      ~Vetor(void);
      int tamanho(void) { return tam; }
      double & operator[](int n) { return mem[n]; }
      Vetor operator=(Vetor v);
   private:
      int tam;
      int *mem;
};
Vetor::Vetor(int n)
{
   tam = n;
   mem = new double[n];
}
Vetor::Vetor(const Vetor &v)
{
   tam = v.tam;
   mem = new double[tam];
   for(int i=0; i<tam; i++)
         mem[i] = v.mem[i];
}
Vetor::~Vetor(void)
{
   delete[] mem;
}
Vetor Vetor::operator=(Vetor v)
{
   if( this != &v ) {
      delete[] mem;
      tam = v.tam;
      mem = new double[v.tam];
      for(int i=0; i<v.tamanho(); i++)
          mem[i] = v.mem[i];
   }
   return *this;
}
```

Essa classe Vetor, apesar de ser bastante simples, proporciona facilidades muito interessantes para se trabalhar com vetores:

- mantém o tamanho como parte da própria estrutura do vetor;
- permite a atribuição, e redimensionamento, entre variáveis do tipo vetor;
- proporciona a mesma sintaxe básica usada com vetores comuns da linguagem C++.

O exemplo a seguir mostra o uso da classe Vetor na codificação de um programa que obtém as temperaturas diárias, registradas durante uma semana, e exibe a temperatura média para o período.

Exemplo 6.4:

```
void le(Vetor &v)
{
   for(int i=0; i<v.tamanho(); i++) {
      cout << i << "º valor? ";
      cin >> v[i];
   }
}
double media(Vetor &v)
{
   double soma=0;
   for(int i=0; i<v.tamanho(); i++)
      soma += v[i];
   return soma/v.tamanho();
}
void main(void)
{
   Vetor temperaturas(7);
   le(temperaturas);
   cout << "Media: " << media(temperaturas) << endl;
}
```

O próximo exemplo mostra a derivação da classe VetLim, em que a indexação é verificada em tempo de execução. Caso seja usado um índice fora do intervalo permitido, uma mensagem de erro será exibida e o programa abortado. Essa classe representará vetores da forma v[p..u], sendo p índice do primeiro elemento e u e índice do ultimo.

Exemplo 6.5:

```
class VetLim : public Vetor {
   public:
      VetLim(int p,int u) : Vetor(u-p+1) { pri=p; ult=u; }
      int primeiro(void) { return pri; }
      int ultimo(void) { return ult; }
      double & operator[](int);
      Vetor operator=(Vetor v);
   private:
      int pri;
      int ult;
};
```

O método operator[] terá que ser redefinido na classe derivada e, como o acesso aos elementos é privativo à classe base, o método herdado terá que ser usado:

Exemplo 6.6:

```
int & VetLim::operator[](int n) // método redefinido para a classe VetLim
{
   if( n<pri || n> ult ) {
      cerr << "Erro: indice fora do intervalo: " << pri << ".." << ult << endl;
      exit(0);
   }
   return Vetor::operator[](n-pri); // o acesso é feito usando o método herdado
}
```

6.4. Declaração de acesso

Independentemente do tipo de herança, protegida ou privada, podemos especificar que certos membros da classe base deverão conservar seu modo de acesso na classe derivada. Esse mecanismo, denominado *declaração de acesso*, não permite de modo algum aumentar ou diminuir a visibilidade de um membro da classe base.

Exemplo 6.7:

```
class X {
   public:
      void m1();
      void m2();
   protected:
      void m3();
      void m4();
};
class Y : private X {
   public:
      X::m1();  // m1() continua público na classe derivada
      X::m3();  // erro: um membro protegido não pode se tornar público
   protected:
      X::m4();  // m4() continua protegido na classe derivada
      X::m2();  // erro: um membro público não pode se tornar protegido
};
```

6.5. Herança, construtores e destrutores

Os construtores, construtores de cópia, destrutores e operadores de atribuição não devem ser herdados, já que uma classe derivada pode ter mais dados membros que sua classe base. Os construtores *default* das classes bases são automaticamente chamados antes dos construtores das classes derivadas. Para não chamar um construtor *default*, e sim um construtor com parâmetros, devemos usar uma lista de inicialização. A chamada de destrutores ocorre na ordem inversa àquela dos construtores.

Como ilustração, vamos derivar uma classe Taxi a partir de uma classe Veiculo. Essa última possuirá dados comuns a todos os veículos, o modelo e a placa. Já a classe derivada terá como dado membro o número de licença de circulação, necessário apenas para taxis.

Exemplo 6.8:

```
class Veiculo {
   public:
      Veiculo(char *modelo, char *placa);
   private:
      char *modelo;
      char *placa;
};
class Taxi : public Veiculo {
   public:
      Taxi(char *modelo, char *placa, int licenca);
   private:
      int licenca;
};
```

Quando um objeto da classe Taxi é criado, primeiro é chamado o construtor da classe Veiculo e só depois é que o seu construtor é executado. Os dados membros modelo e placa são privativos da classe base e, portanto, são inacessíveis à classe derivada. Então, para inicializar esses dados, devemos usar uma lista de inicialização na definição do construtor da classe Taxi:

Exemplo 6.9:

```
Taxi::Taxi(char *modelo, char *placa, int licenca) : Veiculo(modelo,placa)
{
   this->licenca = licenca;
}
```

Agora, para constatar que os construtores das classes bases são realmente chamados antes dos construtores das classes derivadas, e que os destrutores são chamados na ordem inversa, vamos ver mais um exemplo:

Exemplo 6.10:

```
#include <iostream.h>
class Animal {
   public:
      Animal() { cout << "- construindo animal" << endl; }
     ~Animal() { cout << "- destruindo animal" << endl; }
};
class Mamifero : public Animal {
   public:
      Mamifero() { cout << "-- construindo mamifero" << endl; }
     ~Mamifero() { cout << "-- destruindo mamifero" << endl; }
};
class Homem : public Mamifero {
   public:
      Homem() { cout << "--- construindo homem" << endl; }
     ~Homem() { cout << "--- destruindo homem" << endl; }
};
void main(void)
{
   Homem h;
}
```

Quando executado, esse programa produz a seguinte saída:

```
- construindo animal
-- construindo mamifero
--- construindo homem
--- destruindo homem
-- destruindo mamifero
- destruindo animal
```

6.6. Conversão de tipo numa hierarquia de classes

É possível converter implicitamente uma instância de uma classe derivada numa instância de sua classe base, se a herança for pública. O inverso não é permitido, pois o compilador não teria como inicializar os dados membros adicionais da classe derivada.

Exemplo 6.11 :

```
class Veiculo {
   public:
      void exibePlaca() { cout << this->placa << endl; }
      ...
};
class Taxi : public Veiculo {
   ...
};
void manipula(Veiculo v) // manipula qualquer tipo de veículo!
{
   ...
   v.exibePlaca();
   ...
}
void main()
{
   Taxi t;
   manipula( t ); // OK, taxi é um tipo de veículo!
}
```

Da mesma forma, podemos usar ponteiros. Caso a herança seja pública, um ponteiro (ou uma referência) para um objeto de uma classe derivada pode ser implicitamente convertido em um ponteiro (ou uma referência) para um objeto da classe base. É o tipo do ponteiro que determina o método a ser chamado.

Exemplo 6.12 :

```
class Veiculo {
   public:
      void exibePlaca() { cout << this->placa << endl; }
      ...
};
class Taxi : public Veiculo {
   public:
      void exibePlaca() { cout << "VERMELHA: "; Veiculo::exibePlaca(); }
      ...
};
```

```
void manipula(Veiculo v) { // manipula qualquer tipo de veículo!
  ...
  v.exibePlaca();
  ...
}
void main() {
   Taxi *t = new Taxi("GOL","CKX-9265",52634);
   Veiculo *v = t;
   t->exibePlaca();  // exibe placa de taxi
   v->exibePlaca();  // exibe placa de um veículo qualquer
}
```

A execução do programa produzirá a seguinte saída:

```
VERMELHA: CKX-9265
CKX-9265
```

6.7. Herança múltipla

Em C++ é possível usar herança múltipla. Esse tipo de herança permite criar classes derivadas a partir de diversas classes bases, sendo que para cada classe base é preciso definir um modo de herança específico. Se um mesmo nome de membro for usado em duas ou mais classes bases, então será preciso empregar o operador de resolução de escopo para resolver o conflito:

Exemplo 6.13 :

```
class A {
   public:
      void ma() { ... }
   protected:
      int x;
};
class B {
   public:
      void mb() { ... }
   protected:
      int x;
};
class C: public B, public A {
   public:
      void mc();
};
void C::mc()
{
   int i;
   ma();
   mb();
   i = A::x + B::x; // resolução de conflito
   f(i);
}
```

Numa herança múltipla, os construtores são chamados na mesma ordem de declaração de herança. No exemplo a seguir, o construtor *default* da classe C chama o construtor *default* da classe B seguido do construtor *default* da classe A e, só então, o construtor da classe derivada é executado, mesmo que exista uma lista de inicialização.

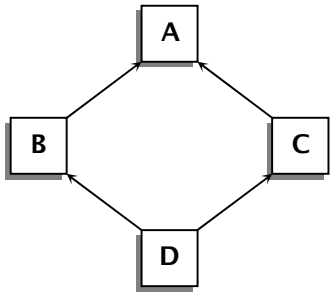
Exemplo 6.14:

```
class A { ... };
class B { ... };
class C: public B, public A { // ordem de chamada dos construtores!
   public:
      C(int i, int j) : A(i) , B(j) { ... }
   ...
};
void main()
{
  C c; // chama os construtores B(), A() et C(), nessa ordem!
  ...
}
```

Na herança múltipla, os destrutores também são chamados na ordem inversa àquela em que os construtores são executados.

6.8. Herança virtual

Considere a derivação da classe D ilustrada a seguir:



Se a classe A possui um dado membro a, um objeto da classe D herdará duas cópias desse dado, uma vez da classe B e outra da classe C. Para acessar esses dados duplicados, será preciso empregar o operador de resolução de escopo.

Exemplo 6.15:

```
void main(void)
{
  D d;
  d.a = 0;    // erro: ambíguo
  d.B::a = 1; // Ok, acessa cópia herdada de B
  d.C::a = 2; // Ok, acessa cópia herdade de C
}
```

Através da *herança virtual*, entretanto, é possível especificar que haja apenas uma ocorrência dos dados herdados da classe base. Para que a classe D herde apenas uma cópia dos dados da classe A, é preciso que as classes B e C herdem virtualmente de A.

Exemplo 6.16:

```
class B : virtual public A   { ... };
class C : virtual public A   { ... };
class D : public B, public C { ... };
void main()
{
  D d;
  d.a = 0; // Ok, não há mais ambigüidade
}
```

É preciso não confundir esse *status* "virtual" da declaração de herança de classes com aquele dos membros virtuais que veremos mais adiante. Aqui, a palavra chave virtual apenas indica ao compilador que os dados herdados não deverão ser duplicados.

6.9. Polimorfismo e funções virtuais

A herança nos permite reutilizar numa classe derivada um código escrito para uma classe base. Nesse processo, alguns métodos podem ser redefinidos e, embora tenham nomes idênticos, ter implementações completamente distintas.

Graças ao *polimorfismo*, podemos usar uma mesma instrução para chamar, dinamicamente, métodos completamente distintos numa hierarquia de classes. Em C++, o polimorfismo é obtido através de *funções virtuais*.

Se uma classe base tem uma função virtual f() e sua classe derivada D também tem uma função f(), do mesmo tipo, então uma chamada a f() a partir de um objeto da classe derivada estará invocando D::f(); mesmo que o acesso seja feito através de um ponteiro ou de uma referência. A função da classe derivada *sobrescreve* a função da classe base; mas, se os tipos das funções forem diferentes, as funções são consideradas distintas e o mecanismo virtual não é empregado.

Exemplo 6.17:

```
class Componente {
   public:
      void exibe() const { cout << "Componente::exibe()" << endl; }
};
class Botao: public Componente {
   public:
      void exibe() const { cout << "Botao::exibe()" << endl; }
};
class Janela: public Componente {
   public:
      void exibe() const { cout << "Janela::exibe()" << endl; }
};
void manipula(const Componente &c)
{
   c.exibe();
}
```

```
void main()
{
  Botao ok;
  Janela ajuda;
  manipula(ok);
  manipula(ajuda);
}
```

Executando o programa acima, veremos que a instrução c.exibe(), na função de manipulação, chamará o método exibe() da classe Componente. Mas isso, na prática, seria um erro; pois a exibição de um botão é uma tarefa diferente da exibição de uma janela. Se na função de manipulação queremos chamar o método exibe() correspondente à classe a que pertence o objeto passado à função, então temos que definir, dentro da classe base, o método exibe() como virtual.

Exemplo 6.18:

```
class Componente {
   public:
      virtual void exibe() const { cout << "Componente::exibe()" << endl; }
};
```

Para maior clareza, a palavra chave virtual pode ser repetida também na declaração do método exibe() nas classes derivadas.

Exemplo 6.19:

```
class Botao: public Componente {
   public:
      virtual void exibe() const { cout << "Botao::exibe()" << endl; }
};
class Janela: public Componente {
   public:
      virtual void exibe() const { cout << "Janela::exibe()" << endl; }
};
```

Ao encontrar uma chamada a um método virtual, o compilador terá que aguardar até o momento da execução para decidir qual o método correto a ser chamado.

Quando trabalhamos com ponteiros para objetos de classes derivadas, é importante que os destrutores também sejam declarados virtuais. Se um objeto da classe derivada está sendo apontado por ponteiro da classe base, e os destrutores são virtuais, então, quando esse objeto é liberado, primeiro será executado o destrutor da classe derivada e só depois o da classe base.

Exemplo 6.20:

```
class Componente {
   public:
      virtual ~Componente() { cout << "Destrói componente" << endl; }
};
class Botao: public Componente {
   public:
      ~Botao() { cout << "Destrói botao" << endl; }
};
```

```
void main(void)
{
   Botao *ok = new Botao;
   Componente *c = ok;
   delete c;
}
```

Ao ser executado o comando delete, no programa acima, serão exibidas as mensagens "Destrói botão" e "Destrói componente". Porém, se o destrutor não tivesse sido declarado virtual, apenas o destrutor da classe Componente, aquela a que pertence o ponteiro, seria chamado.

Nem construtores nem métodos estáticos podem ser declarados virtuais. Além disso, a acessibilidade de um método virtual é conservada dentro de todas as classes derivadas, mesmo que ele seja redefinido com um status diferente .

6.10. Classes abstratas

Às vezes, um método virtual definido em uma classe base, serve como uma interface genérica, que deverá ser implementada nas classes derivadas. Se não há implementação desse método na classe base, dizemos que tal método é *virtual puro*. Para indicar que um método é virtual puro, adicionamos o sufixo = 0 ao seu protótipo.

Exemplo 6.21:

```
class Componente {
  public:
    virtual void exibe() const = 0; // método virtual puro
};
void main()
{
   Componente c;  // erro, não se pode instanciar uma classe abstrata!
   ...
}
```

Uma classe é *abstrata* se contém pelo menos um método virtual puro. Não é possível criar uma instância de uma classe abstrata. Além disso, uma classe abstrata não pode ser usada como tipo de argumento nem como tipo de retorno de uma função.

Um método virtual puro não precisa ter uma implementação. Além disso, uma classe abstrata somente pode ser usada a partir de um ponteiro ou de uma referência. Uma classe derivada que não redefine um método virtual é também considerada abstrata.

7. Fluxos

Esse capítulo descreve o sistema de E/S em fluxos, que permite leitura e escrita tanto de tipos primitivos quanto de tipos definidos, de maneira simples, segura e eficiente.

7.1. Generalidades

Um *fluxo*, ou *stream*, é uma abstração representando a transmissão de dados entre:

- um emissor, aquele que produz informação;
- um receptor, aquele que consome a informação produzida.

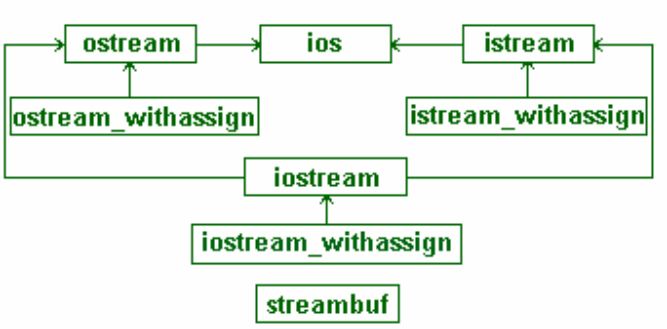
Em C++, as instruções de E/S não fazem parte da linguagem. Elas são oferecidas em uma biblioteca padrão, que implementa os fluxos a partir de uma hierarquia de classes. Por *default*, cada programa C++ pode utilizar três fluxos:

- cout: que corresponde à saída padrão.
- cin: que corresponde à entrada padrão.
- cerr: que corresponde à saída padrão de erros.

Para utilizar outros fluxos, devemos criá-los e associá-los a arquivos, normais ou especiais (dispositivos), ou então a vetores de caracteres.

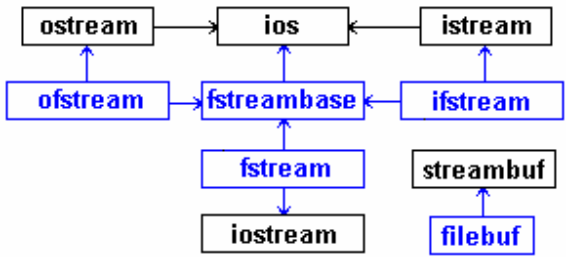
7.2. Fluxos e classes

As classes declaradas em iostream.h permitem a manipulação dos periféricos padrões, como por exemplo vídeo e teclado:



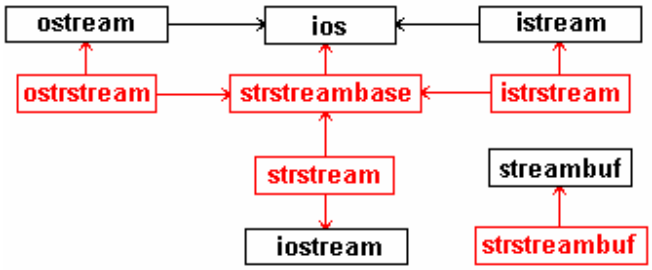
- ios: classe base de E/S em fluxos; ela contém um objeto da classe streambuf para a gestão dos *buffers* de entrada e saída..
- istream : classe derivada de ios para os fluxos de entrada.
- ostream : classe derivada de ios para os fluxos de saída.
- iostream : classe derivada de istream e de ostream para os fluxos bidirecionais.
- istream_withassign, ostream_withassign e iostream_withassign: classes derivadas respectivamente de istream, ostream e iostream, que oferecem o operador de atribuição. Os fluxos padrões cin, cout et cerr são instâncias dessas classes.

As classes declaradas em fstream.h permitem a manipulação de arquivos em disco:



- fstreambase : classe base para as classes derivadas ifstream, ofstream e fstream. Ela mesma é derivada de ios e contém um objeto da classe filebuf.
- ifstream : classe que permite realizar entrada a partir de arquivos.
- ofstream : classe que permite efetuar saída em arquivos.
- fstream : classe que permite realizar entrada/saída em arquivos.

As classes declaradas em strstream.h permitem simular operações de entrada e saída a partir de *buffers* na memória principal. Elas operam da mesma maneira que as funções sprintf() e sscanf() da linguagem C.



- strtreambase : classe base para as classes seguinte. Ela contém um objeto da classe strstreambuf (derivada de streambuf).
- istrstream : derivada de strtreambase e de istream, permite leitura a partir de um *buffer*, da mesma maneira que a função sscanf().
- ostrstream : derivada de strtreambase e de ostream, permite escrita em *buffer*, do mesmo modo que a função sprintf().
- strstream : derivada de istrstream e de iostream, permite leitura e escrita em *buffer*.

7.3. O fluxo de saída padrão

O fluxo de saída padrão, ostream, permite:

- saída formatada e não formatada (em um *streambuf*)
- sobrecarga do operador de inserção <<.
- exibir tipos predefinidos da linguagem C++.
- exibir tipos definidos pelo programador.

O exemplo a seguir ilustra a sobrecarga do operador de inserção em fluxo (>>) para a classe de números complexos, vista anteriormente:

Exemplo 7.1:

```
#include <iostream.h>
#include <math.h>
class Complexo {
   public:
      Complexo(double a, double b) { r=a; i=b; }
      Complexo operator+(Complexo c) { return Complexo(r+c.r,i+c.i); }
      friend ostream & operator<<(ostream &out,const Complexo &x) {
         return out << x.r << (x.i<0 ? "-" : "+") << fabs(x.i) << "i";
      }
   private:
      double r; // parte real
      double i; // parte imaginária
};
void main(void)
{
   Complexo a(1,2), b(3,-4);
   cout << a+b << endl;
}
```

A função operator<< retorna uma referência ao objeto ostream para o qual foi chamada. Isso permite chamá-la, de forma encadeada, várias vezes numa mesma instrução.

Exemplo 7.2:

```
void main(void)
{
   Complexo a(1,2), b(3,-4);
   cout << a << " + " << b << " = " << a+b << endl;
}
```

Além do operador de inserção (<<), a classe ostream possui os seguintes métodos:

ostream & put(char c)	insere um caracter num fluxo
ostream & write(const char *, int n)	insere *n* caracteres num fluxo
streampos tellp()	retorna a posição corrente dentro do fluxo
ostream & seekp(streampos n)	posiciona-se a *n* bytes a partir do início do arquivo. O tipo streampos corresponde à uma posição do arquivo, que inicia-se em 0.
ostream & seekp(streamoff d, seek_dir r)	Posiciona-se a *d* bytes a partir do início do arquivo (*r* = beg), a posição corrente (*r* = cur) ou o final do fluxo (*r* = end)
ostream & flush()	esvazia o *buffer* do fluxo

Exemplo 7.3:

```
#include <ostream.h>
void main(void)
{
  cout.write("Hello world!", 12);
  cout.put('\n');
}
```

7.4. O fluxo de entrada padrão

O fluxo de entrada padrão, istream, permite:

- entrada formatada e não formatada (em um *streambuf*)
- sobrecarga do operador de extração >>.
- entrada de tipos predefinidos da linguagem C++.
- entrada de tipos definidos pelo programador.

O próximo exemplo mostra a sobrecarga do operador >> para a classe Complexo:

Exemplo 7.4:

```
class Complexo {
    friend istream & operator>>(istream &in, Complexo &x);
    ...
};
```

Assim como no caso de inserção, para que o operador de extração possa ser usado de maneira encadeada, a função operator>> também deve retornar uma referência ao objeto istream a partir do qual ela foi chamada.

Exemplo 7.5 :

```
istream operator>> istream & operator>>(istream &in, Complexo &x)
{
   in >> x.r;
   char sinal;
   in >> sinal;
   assert( sinal=='+' || sinal=='-' );
   in >> x.i;
   if( sinal=='-' ) x.i *= -1;
   char i;
   in >> i;
   if( i!='i' ) x.i=0;
   return in;
}
void main(void)
{
   Complexo x;
   cout << "Digite um número complexo: ";
   cin >> x;
   ...
}
```

Além do operador de extração, a classe istream oferece os seguintes métodos:

int get()	extrai e retorna um caracter (ou EOF)
istream & get(char &c)	extrai e armazena em *c* um caracter
int peek()	informa o próximo caracter, sem extraí-lo
istream & get(char *ch, int n, char d='\n')	extrai *n-1* caracteres do fluxo e os armazena a partir do endereço *ch*, parando assim que o delimitador é encontrado.
Istream & getline(char *ch, int n, char d='\n')	como o método anterior, exceto que o delimitador é extraído e descartado
istream & read(char *ch, int n)	extrai um bloco de pelo menos *n* bytes do fluxo e os armazena a partir do endereço *ch*. O número de bytes efetivamente lidos é obtido através do método gcount()
int gcount()	retorna o número de caracteres extraídos na última leitura
streampos tellg()	retorna a posição corrente dentro do fluxo
istream & seekg(streampos n)	posiciona-se a *n* bytes a partir do início do arquivo. O tipo streampos corresponde à uma posição do arquivo, iniciando em 0.
istream & seekg(streamoff d, seek_dir r)	Posiciona-se a *d* bytes a partir do início do arquivo (*r* = beg), a posição corrente (*r* = cur) ou o final do fluxo (*r* = end)
istream & flush()	esvazia o *buffer* do fluxo

7.5. Controle de estado de um fluxo

A classe ios descreve os aspectos comuns aos fluxos de entrada e saída. É uma classe base virtual para todos os fluxos e, portanto, não podemos jamais criar um objeto dessa classe. Porém, podemos utilizar seus métodos para testar o estado de um fluxo ou para controlar o formato das informações. Além disso, essa classe oferece uma série de constantes muito úteis na manipulação de fluxos.

A classe ios oferece os seguintes métodos:

int good()	retorna um valor diferente de zero se a última operação de E/S teve sucesso
int fail()	faz o inverso do método anterior
int eof()	retorna um valor diferente de zero se o final do arquivo foi atingido
int bad()	retorne um valor diferente de zero se uma operação foi malsucedida
int rdstate()	retorna o valor da variável de estado do fluxo ou zero se tudo estiver bem
void clear()	zera o indicador de erros do fluxo

7.6. Associando um fluxo a um arquivo

É possível criar um fluxo associado a um arquivo em disco. As três classes que permitem acessar arquivos em disco são definidas em fstream.h:

- ifstream: permite acessar arquivos para leitura.
- ofstream: permite acessar arquivos para gravação.
- fstream: permite acessar arquivos para leitura e gravação.

Cada uma dessas classes usa um *buffer* da classe filebuf para sincronizar as operações entre o fluxo e o disco. A classe fstreambase oferece uma coleção de métodos comuns a essas três classes; em particular, o método open(), que abre um arquivo em disco e o associa a um fluxo:

```
void open(const char *nome, int modo, int prot=filebuf::openprot);
```

Nessa sintaxe, nome refere-se ao arquivo que será associado ao fluxo, modo indica o modo de seu abertura e prot define os direitos de acesso ao arquivo (por *default*, os direitos de acesso são estabelecidos pelo sistema operacional).

O modo de abertura deve ser um dos elementos da enumeração a seguir:

```
enum open_mode {
   app,       // anexa os dados ao final do arquivo
   ate,       // posiciona-se no final do arquivo
   in,        // abre para leitura (é o default para ifstream)
   out,       // abre para gravação (é o default para ofstream)
   binary,    // abre em modo binário (o default é o modo texto)
   trunc,     // destrói o arquivo se ele existe e o recria
   nocreate, // se o arquivo não existe, a abertura falha
   noreplace // se o arquivo existe, a abertura falha
};
```

Para a classe ifstream o modo de abertura default é ios::in e para a classe ofstream, é ios::out.

Exemplo 7.6:

```
#include <fstream.h>

ifstream in;
in.open("teste.tmp");                          // abre para leitura, por default
ofstream out;
out.open("teste.tmp");                         // abre para gravação, por default
fstream io;
io.open("teste.tmp", ios::in | ios::out);      // abre para leitura e gravação
```

Podemos também chamar os construtores dessas três classes e combinar as operações de criação e abertura de fluxo numa única instrução:

Exemplo 7.7:

```
#include <fstream.h>

ifstream in("teste.tmp");                      // abre para leitura, por default
ofstream out("teste.tmp");                     // abre para gravação, por default
fstream io("teste.tmp", ios::in | ios::out);   // abre para leitura e gravação
```

7.7. Formatação de dados

Cada fluxo conserva permanentemente um conjunto de indicadores especificando a formatação corrente. Isso permite dar um comportamento *default* ao fluxo, ao contrário do que ocorre com as funções printf() e scanf() da linguagem C, que exigem a indicação de um formato para cada operação de E/S realizada.

O indicador de formato do fluxo é um inteiro longo (protegido) definido na classe ios:

```
class ios {
   public:
   // ...
      enum {
         skipws,     // ignora os espaços na entrada
         left,       // justifica as saídas à esquerda
         right,      // justifica as saídas à direita
         internal,   // preenche entre o sinal e o valor
         dec,        // conversão em decimal
         oct,        // conversão em octal
         hex,        // conversão em hexadecimal
         showbase,   // exibe o indicador de base
         showpoint,  // exibe ponto decimal nos números reais
         uppercase,  // exibe hexadecimais em maiúsculas
         showpos,    // exibe sinal em números positivos
         scientific, // notação científica (1.234000E2) para reais
         fixed,      // notação fixa (123.4) para reais
         unitbuf,    // esvazia o fluxo após uma inserção
         stdio       // permite utilizar stdout et cout
      };
   //...
   protected:
      long x_flags; // indicador de formato
   //...
};
```

A classe ios também define constantes através das quais acessamos os indicadores:

- static const long basefield : permite definir a base (*dec*, *oct* ou *hex*)
- static const long adjustfield :permite definir o alinhamento (*left*, *right* ou *internal*)
- static const long floatfield :permite definir a notação para reais (*scientific* ou *fixed*)

Os métodos seguintes (também definidos em ios) permitem ler ou modificar os valores dos indicadores de formato:

long flags()	retorna o valor do indicador de formato
long flags(long f)	modifica os indicadores com o valor de f
long setf(long setbits, long field)	altera somente os bits do campo indicado por field
long setf(long f)	modifica o valor do indicador de formato
long unsetf(long)	zera o valor do indicador de formato

Exemplo 7.8:

```
#include <iostream.h>
#include <iomanip.h>
void main(void)
{
   cout.setf(ios::dec,ios::basefield);
   cout << 255 << endl;
   cout.setf(ios::oct,ios::basefield);
   cout << 255 << endl;
   cout.setf(ios::hex,ios::basefield);
   cout << 255 << endl;
}
```

A execução desse programa produzirá a seguinte saída:

```
255
377
ff
```

Os métodos a seguir permite definir tamanho de campo e caracter de preenchimento:

Int width(int)	define a largura do campo para a próxima saída
Int width()	devolve a largura do campo de saída
char fill(char)	define o caracter de preenchimento de campo
char fill()	devolve o caracter de preenchimento de campo
Int precision(int)	define o número de caracteres (menos o ponto) que um real ocupa
Int precision()	devolve o número de caracteres (menos o ponto) que um real

	ocupa

Exemplo 7.9:

```
#include <iostream.h>
#include <iomanip.h>
void main(void)
{
   cout << "(";
   cout.width(4);
   cout.setf(ios::right,ios::adjustfield);
   cout << -45 << ")" << endl;
   cout << "(";
   cout.width(4);
   cout.setf(ios::left,ios::adjustfield);
   cout << -45 << ")" << endl;
   cout << "(";
   cout.width(4);
   cout.setf(ios::internal,ios::adjustfield);
   cout << -45 << ")" << endl;
}
```

Esse outro exibirá o seguinte:

```
( -45)
(-45 )
(- 45)
```

Exemplo 7.10:

```
#include <iostream.h>
#include <iomanip.h>
void main(void)
{
   cout.setf(ios::scientific,ios::floatfield);
   cout << 1234.56789 << endl;
   cout.precision(2);
   cout.setf(ios::fixed,ios::floatfield);
   cout << 1234.56789 << endl;
}
```

A saída exibe o valor em notação científica e em notação fixa:

```
1.2345678e+03
1234.57
```

7.8. Os manipuladores

O uso dos métodos de formatação leva a instruções um tanto longas. Por outro lado, usando manipuladores podemos escrever um código mais compacto e mais legível. Em vez de escrever:

```
cout.width(10);
cout.fill('*');
cout.setf(ios::hex,ios::basefield);
cout << 123 ;
cout.flush();
```

vamos preferir a instrução equivalente:

```
cout << setw(10) << setfill('*') << hex << 123 << flush;
```

As classes ios, istream e ostream implementam os manipuladores predefinidos. O arquivo de inclusão iomanip.h define um certo número desses manipuladores:

dec	a próxima operação de E/S usa base decimal
oct	a próxima operação de E/S usa base octal
hex	a próxima operação de E/S usa base hexadecimal
endl	escreve '\n' e depois descarrega o *buffer*
ends	escreve '\0' e depois descarrega o *buffer*
flush	descarrega o *buffer*
ws	descarta os espaços num fluxo de entrada
setbase(int b)	define a base para a próxima saída
setfill(int c)	define o caracter de preenchimento de campo
setprecision(int p)	define o número de dígitos na próxima saída de número real
setw(int l)	define a largura do campo para a próxima saída
setiosflags(long n)	ativa os bits do indicador de formato especificado por *n*
resetiosflags(long b)	desativa os bits do indicador de formato indicado por *b*

Podemos também escrever nossos próprios manipuladores. Por exemplo, um dos manipuladores mais utilizados, endl, é definido da seguinte maneira:

Exemplo 7.11 :

```
ostream &flush(ostream &os) { return os.flush(); }
ostream &endl(ostream &os) { return os << '\n' << flush; }
```

Para utilizá-lo numa instrução do tipo:

```
cout << endl;
```

a função *operator<<* é sobrecarregada na classe ostream como:

Exemplo 7.12 :

```
ostream &ostream::operator<<(ostream& (*f)(ostream &))
{
   (*f)(*this);
   return *this;
}
```

A função operator<< acima recebe como parâmetro um ponteiro para a função que implementa o manipulador a ser executado. Através desse ponteiro a função que

implementa o manipulador é chamada e, depois, o objeto ostream é devolvido para que o operador possa ser usado de forma encadeada.

Vamos implementar, como exemplo, um manipulador para tabulação:

Exemplo 7.13:

```
ostream &tab(ostream &os)
{
   return os << '\t';
}
```

Esse manipulador poderá ser usado assim:

```
cout << 12 << tab << 34;
```

7.9. Criando manipuladores

A implementação de um novo manipulador consiste em duas partes:

- *manipulador*: sua forma geral para ostream é:

  ```
  ostream &<manipulador>(ostream &,<tipo>);
  ```

  sendo <tipo> o tipo do parâmetro do manipulador.

  Essa função não pode ser chamada diretamente por uma instrução de E/S, ela será chamada somente pelo aplicador.

- *aplicador*: ele chama o manipulador. É uma função global cuja forma geral é:

  ```
  <x>MANIP( <tipo> )<manipulador>(<tipo> <arg>)
  {
     return <x>MANIP(<tipo>) (<manipulador>,<arg>);
  }
  ```

  com <x> valendo O para manipuladores de ostream (e seus derivados), I, S e IO para, respectivamente, para manipuladores de istream, ios e iostream.

Como exemplo, vamos criar um manipulador para exibir um número em binário:

Exemplo 7.14:

```
#include <iomanip.h>
#include <limits.h>  // ULONG_MAX
ostream &bin(ostream &os, long val)
{
   unsigned long mascara = ~(ULONG_MAX >> 1);
   while ( mascara ) {
      os << ((val & mascara) ? '1' : '0');
      mascara >>= 1;
   }
   return os;
}
OMANIP(long) bin(long val)
{
   return OMANIP(long) (bin, val);
}
```

```
void main()
{
  cout << "2000 em binário é " << bin(2000) << endl;
}
```

8. Exceções

Esse capítulo apresenta o mecanismo de manipulação de exceções e algumas técnicas de tratamento de erros que ele suporta. Esse mecanismo é baseado em expressões que lançam exceções que são depois capturadas por manipuladores.

8.1. A estrutura *try...catch*

Uma *exceção* ocorre quando um programa encontra-se numa situação anormal, devido a condições que estão fora de seu controle, tais como memória insuficiente, erros de E/S, índice fora do intervalo permitido, etc. Nessas ocasiões, a solução usualmente adotada consiste em exibir uma mensagem de erro e interromper a execução do programa, devolvendo um código de erro.

O fragmento da classe Vetor, a seguir, ilustra essa idéia:

Exemplo 8.1:

```
class Vetor {
   public:
      Vetor(int n);
      int & operator[](int n);
      ...
   private:
      int tam;
      int *mem;
};
Vetor::Vetor(int n)
{
   if( n<=0 ) {
      cout << "tamanho de vetor invalido: " << n << endl;
      exit(1);
   }
   tam = n;
   mem = new int[n];
}
int & operator[](int n)
{
   if( n<0 || n>=tam ) {
      cout << "índice fora do intervalo: " << n << endl;
      exit(2);
   }
   return mem[n];
}
```

Em C++, entretanto, podemos usar exceções para gerenciar de forma mais eficaz os erros que ocorrem durante a execução de um programa. Em vez de simplesmente abortar sua execução, podemos comunicar a ocorrência do evento inesperado a um contexto de execução mais alto, que esteja melhor capacitado para se recuperar da condição de erro.

Para lançar uma exceção a um nível mais alto de execução, usamos o comando throw:

Exemplo 8.2:

```
Vetor::Vetor(int n)
{
   if( n<=0 ) throw 1;
   tam = n;
   mem = new int[n];
}
int & operator[](int n)
{
   if( n<0 || n>=tam ) throw 2;
   return mem[n];
}
```

Uma exceção lançada é tratada depois, por um código que se encontra fora do fluxo de execução normal do programa. Para tratar uma exceção, usamos a seguinte estrutura:

```
try { ... }
catch( ... ) { ... }
```

Quando uma exceção é detectada dentro de um bloco try, os destrutores dos objetos criados no bloco são chamados e, em seguida, o controle é transferido ao bloco catch correspondente ao tipo da exceção detectada, se ele existir. Ao final da execução do bloco catch, o programa continua sua execução normalmente, na primeira instrução após a estrutura try...catch.

Um bloco try deve ser seguido de pelo menos um bloco catch. Além disso, se existirem vários deles, cada um deve interceptar um tipo de exceção diferente.

Exemplo 8.3:

```
...
try {
  int n;
  cout << "Qual o tamanho do vetor? ";
  cin >> n;
  Vetor v(n);
  int p;
  cout << "Qual a posição? ";
  cin >> p;
  cout << "Qual o valor a ser armazenado? ";
  cin >> v[p];
}
```

```
catch(int n) {
   switch( n ) {
      case 1: cout << "tamanho inválido" << endl; break;
      case 2: cout << "índice inválido" << endl; break;
   }
}
...
```

8.2. Discriminando exceções

O bloco try é uma seção segura do código. Se uma exceção é lançada enquanto ele está executando, um bloco catch apropriado, denominado *manipulador* da exceção, é selecionado para assumir o controle da execução. Essa seleção é feita com base da declaração que aparece entre parênteses, na frente da palavra reservada catch:

- catch (T): manipula exceções do tipo T e de todas as suas classes derivadas.
- catch (T e): idem; mas dispõe o objeto e contendo informações sobre a exceção.
- catch (...): manipula exceções de quaisquer tipos.

Quando uma exceção é detectada, o mecanismo seleciona como manipulador o primeiro bloco catch cuja declaração seja do mesmo tipo da exceção. Assim, se a declaração for uma elipse (...),o bloco deve ser o último a ser especificado.

Naturalmente, um programa pode ter vários tipos distintos de erros e podemos representar cada um deles por um número. Entretanto, num enfoque orientado a objetos, seria melhor que cada tipo de exceção fosse representada por uma classe.

Exemplo 8.4 :

```
class Vetor {
   public:
      class ExcecaoDeTamanho {};
      class ExcecaoDeIndexacao {};
      ...
      Vetor(int n);
      int & operator[](int n);
      ...
   private:
      int tam;
      int *mem;
};
Vetor::Vetor(int n)
{
   if( n<=0 ) throw ExcecaoDeTamanho();
   tam = n;
   mem = new int[n];
}
int & operator[](int n)
{
   if( n<0 || n>=tam ) throw ExcecaoDeIndexacao();
   return mem[n];
```

```
}
```

As classes ExcecaoDeTamanho *e* ExcecaoDeIndexacao *estão definidas no escopo da classe* Vetor *e, para acessá-las fora dessa classe, precisamos usar o operador de resolução de escopo (::).*

Agora, o programa que trata a exceção pode ser codificado da seguinte maneira:

Exemplo 8.5:

```
...
try {
  int n;
  cout << "Qual o tamanho do vetor? ";
  cin >> n;
  Vetor v(n);
  int p;
  cout << "Qual a posição? ";
  cin >> p;
  cout << "Qual o valor a ser armazenado? ";
  cin >> v[p];
}
catch(Vetor::ExcecaoDeTamanho) {
  cout << "tamanho inválido" << endl;
}
catch(Vetor::ExcecaoDeIndexacao) {
   cout << "índice inválido" << endl;
}
catch(...) {
   cout << "exceção inesperada" << endl;
}
...
```

No exemplo anterior, uma exceção é capturada através de seu tipo, mas o que é lançado não é um tipo e sim um objeto. Se desejamos transmitir informações do ponto onde a exceção foi lançada ao ponto em que ela será manipulada, então precisamos adicionar dados ao objeto. Por exemplo, pode ser interessante saber o valor do índice que causou uma exceção de indexação.

Exemplo 8.6:

```
class Vetor {
   public:
      class ExcecaoDeIndexacao {
          public:
             int indice;
             ExcecaoDeIndexacao(int n) : indice(n) {}
      };
      ...
      int & operator[](int n);
      ...
   private:
      int tam;
      int *mem;
```

```
};

int & operator[](int n)
{
   if( n<0 || n>=tam ) throw ExcecaoDeIndexacao(n);
   return mem[n];
}
```

O exemplo a seguir mostra como acessar a informação contida numa exceção:

Exemplo 8.7:

```
try {
  cout << "Qual o valor a ser armazenado? ";
  cin >> v[p];
}
catch(Vetor::ExcecaoDeIndexacao e) {
   cout << "índice inválido: " << e.indice << endl;
}
```

8.3. Hierarquia de exceções

Exceções freqüentemente podem ser agrupadas em famílias. Por exemplo, uma família de erros de vetor poderia incluir erros tais como TamanhoInvalido e IndiceInvalido. Uma forma simples de implementar isso seria a seguinte:

```
enum ErroDeVetor { TamanhoInvalido, IndiceInvalido };
```

Entretanto, já vimos que é preferível representar exceções através de classes e, nesse caso, uma solução mais adequada seria usar herança:

Exemplo 8.8:

```
class ExcecaoDeVetor { };
class TamanhoInvalido : public ExcecaoDeVetor { };
class IndiceInvalido  : public ExcecaoDeVetor { };
```

A vantagem dessa hierarquia de exceções é que, se não quisermos tratar cada exceção individualmente, podemos especificar um único manipulador para a classe base e ele, automaticamente, será capaz de capturar todas as exceções de classes derivadas.

Exemplo 8.9:

```
try {
   // usa um vetor
   ...
}
catch(ExcecaoDeVetor) {
   // captura qualquer exceção relativa a vetores
   ...
}
```

A ordem em que os manipuladores aparecem é significativa, pois eles são examinados na mesma ordem em que aparecem e somente o primeiro deles que se mostrar adequado pode ser selecionado.

Como o manipulador de uma classe captura não só as exceções daquela classe, mas também todas as exceções de suas classes derivadas, é considerado um erro colocar o manipulador de uma classe base antes daquele de sua classe derivada. Depois que um manipulador é selecionado, os manipuladores subseqüentes não são mais examinados.

8.4. Declaração das exceções lançadas por uma função

A declaração de exceções, através de uma cláusula throw, nos permite especificar o tipo das exceções que poderão ser eventualmente lançadas por uma função ou método:

Exemplo 8.10:

```
void f1() throw (Erro);
void f2() throw (Erro1, Erro2, Erro3);
void f3() throw ();
void f4();
```

Se nada é declarado, uma função pode lançar qualquer tipo de exceção. No exemplo acima, a função f4() pode lançar qualquer exceção, enquanto a função f3() não pode lançar nenhuma. Já a função f1() pode lançar apenas exceções da classe Erro, ou dela derivada. Se uma função lança uma exceção não especificada em sua declaração, a função unexpected() é chamada.

A cláusula throw deve constar tanto da declaração quanto da definição da função. Além disso, uma função pode *relançar* uma exceção usando um comando throw, sem parâmetro, dentro de um bloco catch. Tais exceções também devem ser declaradas.

A função unexpected() é chamada sempre que uma função lança uma exceção que não foi declarada em sua cláusula throw. Essa função interrompe a execução do programa fazendo uma chamada à função abort(). Podemos definir nossa própria função unexpected() passando seu nome como argumento à função set_unexpected().

Exemplo 8.11 :

```
#include <iostream.h>
#include <stdlib.h>
#include <eh.h>
void inesperado() {
  cout << "exceção inesperada foi lançada" << endl;
  exit(1);
}
class Erro1 {};
class Erro2 {};
class Teste {
  public:
    void e() throw (Erro1);
};
```

```
void Teste::e() throw (Erro1) {
  throw Erro2();
}
void main() {
  try {
     set_unexpected( (unexpected_function) inesperado);
     Teste t;
     t.e();
  }
  catch ( Erro1 ) { ... }
}
```

A função terminate() é chamada sempre que nenhum manipulador correspondente à exceção detectada é encontrado. Essa função interrompe a execução do programa fazendo uma chamada à função abort(). Se for desejado que ela execute alguma outra função antes de terminar o programa, devemos chamar a função set_terminate() passando a ela como argumento o nome da função a ser executada.

O código a seguir lança uma exceção do tipo Erro1, mas apenas exceções do tipo Erro2 podem ser manipuladas. A chamada a set_terminate() instrui terminate() a chamar termina():

Exemplo 8.12 :

```
#include <iostream.h>
#include <stdlib.h>
#include <eh.h>
void termina()
{
   cout << "termina o programa" << endl;
   exit(1);  // não deve retornar a quem chamou
}
class Erro1 {};
class Erro2 {};
class Teste {
  public:
    void e() throw (Erro1);
};
void Teste::e() throw (Erro1) {
  throw Erro1();
}
void main()
{
  try {
     set_terminate((terminate_function) termina);
     Teste t;
     t.e();
  }
  catch ( Erro2 ) { ... }
}
```

Depois de realizar todas as tarefas de finalização, a função termina() deve terminar o programa fazendo uma chamada à função exit(). Se ela não faz isso, e retorna ao ponto onde foi chamada, a função abort() é executada.