Universidade Estadual do Oeste do Paraná - UNIOESTE
Centro de Ciências Exatas e Tecnológicas - CCET
Curso de Ciência da Computação

INTRODUÇÃO À PROGRAMAÇÃO ORIENTADA A OBJETOS EM C++

CASCAVEL - PR
2014

SUMÁRIO

UNIDADE 1 – CONSIDERAÇÕES INICIAIS SOBRE A LINGUAGEM C

1.1 AS ORIGENS DO C/C++

A história do C/C++ começa nos anos 70 com a invenção da linguagem C. A linguagem C foi inventada e implementada pela primeira vez por Dennis Ritchie em um DEC PDP-11, usando o sistema operacional UNIX. A linguagem C é o resultado do processo de desenvolvimento iniciado com outra linguagem, chamada BCPL, desenvolvida por Martin Richards. Esta linguagem influenciou a linguagem inventada por Ken Thompson, chamado B, a qual levou ao desenvolvimento da linguagem C.

A linguagem C tornou-se uma das linguagens de programação mais usadas. Flexível, ainda que poderosa, a linguagem C tem sido utilizada na criação de alguns dos mais importantes produtos de software dos últimos anos. Entretanto, a linguagem encontra seus limites quando o tamanho de um projeto ultrapassa um certo ponto. Ainda que este limite possa variar de projeto para projeto, quanto o tamanho de um programa se encontra entre 25.000 e 100.000 linhas, torna-se problemático o gerenciamento, tendo em vista que é difícil compreende-lo como um todo. Para resolver este problema, em 1980, enquanto trabalhava nos laboratórios da Bell, em Murray Bill, New Jersey, Bjarne Stroustrup acrescentou várias extensões à linguagem C e chamou inicialmente esta nova linguagem de “C com classes”. Entretanto, em 1983, o nome foi mudado para C++.

Muitas adições foram feitas pós-Stroustrup para que a linguagem C pudesse suportar a ***programação orientada a objetos***, às vezes referenciada como ***POO***. Stroustrup declarou que algumas das características da orientação a objetos do C++ foram inspiradas por outra linguagem orientada a objetos chamada de Simula67.

A linguagem C é freqüentemente referenciada como uma linguagem de nível médio, posicionando-se entre o assembler (baixo nível) e o Pascal (alto nível)[1]. Uma das razões da invenção da linguagem C foi dar ao programador uma linguagem de alto nível que poderia ser utilizada como uma substituta para a linguagem assembly.

Como você provavelmente sabe, a linguagem assembly utiliza a representação simbólica das instruções executadas pelo computador. Existe um relacionamento de um para um entre cada instrução da linguagem assembly e a instrução de máquina. Ainda que este relacionamento possibilite escrever programas altamente eficientes, isso torna a programação um tanto quanto tediosa e passível de erro. Por outro lado, linguagens de alto nível, como Pascal, estão extremamente distantes da representação de máquina. Uma instrução em Pascal não possui essencialmente nenhum relacionamento com a seqüência de instruções de máquina que são executadas.

Entretanto, ainda que a linguagem C possua estruturas de controle de alto nível, como é encontrado na Pascal, ela também permite que o programador manipule bits, bytes e endereços de uma maneira mais proximamente ligada à máquina, ao contrário da abstração apresentadas por outras linguagens de alto nível. Por esse motivo, a linguagem C tem sido ocasionalmente chamada de “código assembly de alto nível”. Por sua natureza dupla, a linguagem C permite que sejam criado programas rápidos e eficientes sem a necessidade de se recorrer a linguagem

[1] SCHILDT, H. **Turbo C++**: guia do usuário. São Paulo : Makron Books,1992.

assembly.

A filosofia que existe por trás da linguagem C é que o programador sabe realmente o que está fazendo. Por esse motivo, a linguagem C quase nunca “coloca-se no caminho” do programador, deixando-o livre para usar (ou abusar) dela de qualquer forma que queira. Existe uma pequena verificação de erro de execução ***runtime error***. Por exemplo, se por qualquer motivo você quiser sobrescrever a memória na qual o seu programa está atualmente residindo, o compilador nada fará para impedi-lo. O motivo para essa “liberdade na programação” é permitir ao compilador C criar códigos muito rápidos e eficientes, já que ele deixa a responsabilidade da verificação de erros para você. Em outras palavras, a linguagem C considera que você é hábil o bastante para adicionar suas próprias verificações de erro quando necessário.

Quando C++ foi inventado, Bjarne Stroustrup sabia que era importante manter o espírito original da linguagem C, incluindo a eficiência, a natureza de nível médio e a filosofia de que o programador, não a linguagem, está com as responsabilidades, enquanto, ao mesmo tempo, acrescentava o suporte à programação orientada a objetos. Assim, o C++ proporciona ao programador a liberdade e o controle da linguagem C junto com o poder dos objetos. As características da orientação a objetos em C++, usando as palavras de Stroustrup, “permite aos programas serem estruturados quanto à clareza e extensibilidade, tornando fácil a manutenção sem perda de eficiência”.

UNIDADE 2 – INTRODUÇÃO À PROGRAMAÇÃO ORIENTADA A OBJETOS EM C++

2.1 INTRODUÇÃO

A programação orientada a objetos é uma maneira de abordar a tarefa de programação. Com o decorrer dos anos os programas desenvolvidos tornaram-se por demais complexos; assim, fez-se necessário incorporar mudanças às linguagens de programação frente a esta complexidade crescente.

Nos aos 60 nasceu a programação ***estruturada***. Esse é o método estimulado por linguagens como C e Pascal. Usando-se linguagens estruturadas, foi possível, pela primeira vez, escrever programas moderadamente complexos de maneira razoavelmente fácil. Entretanto, com a programação estruturada, quando um projeto atinge um certo tamanho, ele torna-se incontrolável, pois a sua complexidade excede a capacidade dessa programação. A cada marco no desenvolvimento da programação, métodos foram criados para permitir ao programador tratar com complexidades incrivelmente grandes. Cada passo combinava os melhores elementos dos métodos anteriores com mais avanços. Atualmente, muitos projetos estão próximos ou no ponto em que o tratamento estruturado não mais funciona. Para resolver esse problema, a programação orientada a objetos foi criada.

A programação orientada a objetos aproveitou as melhores idéias da programação estruturada e combinou-as com novos conceitos poderosos para a arte da programação. A programação orientada a objetos permite que um problema seja mais facilmente decomposto em subgrupos relacionados. Então, usando-se a linguagem, pode-se traduzir esses subgrupos em unidades autocontidas chamadas ***objetos***.

Todas as linguagens de programação orientadas a objetos possuem três coisas em comum: ***objetos***, ***polimorfismo*** e ***herança***.

2.1.1 OBJETOS

A característica mais importante de uma linguagem orientada a objetos é o ***objeto***. De maneira simples, um ***objeto*** é uma entidade lógica que contém dados e o código para manipular esses dados. Dentro de um objeto, alguns códigos e/ou dados podem ser privados ao objeto e inacessíveis diretamente para qualquer elemento fora dele. Dessa maneira, um objeto evita significativamente que algumas partes não relacionadas de programa modifiquem ou usem incorretamente as partes privadas do objeto. Essa ligação dos códigos e dos dados é freqüentemente referenciada como ***encapsulação***.

Para todas as intenções e propósitos, um objeto é uma variável de um tipo definido pelo usuário. Pode parecer estranho à primeira vista encarar um objeto, que une tanto código como dados, como sendo uma variável. Contudo, em programação orientada a objetos, esse é precisamente o caso. Quando se define um objetos, cria-se implicitamente um novo tipo de dado.

2.1.2 POLIMORFISMO

Linguagens de programação orientada a objetos suportam ***polimorfismo***, o que significa essencialmente que um nome pode ser usado para muitos propósitos relacionados, mas ligeiramente diferentes. A intenção do polimorfismo é permitir a um nome ser usado para especificar uma classe geral de ações. Entretanto, dependendo do tipo de dado que está sendo tratado, uma instância específica de um caso geral é executada. Por exemplo, pode-se ter um programa que define três tipos diferentes de pilhas. Uma pilha é usada para valores inteiros, uma para valores de ponto flutuante e uma, para inteiros longos. Por causa do polimorfismo, pode-se criar três conjuntos de funções para essas pilhas, chamadas ***push( )*** e ***pop( )***, e o compilador selecionará a rotina correta, dependendo com qual tipo a função é chamada. Nesse exemplo, o conceito geral é pôr e tirar dados de e para a pilha. As funções definem a maneira específica como isso é feito para cada tipo de dado.

As primeiras linguagens de programação orientadas a objetos eram interpretadas; assim, o polimorfismo era, obviamente, suportado no momento da execução. Entretanto, o C++ é uma linguagem compilada. Portanto, em C++, é suportado tanto o polimorfismo em tempo de execução como em tempo de compilação.

2.1.3 HERANÇA

Herança é o processo em que um objeto pode adquirir as propriedades de outro objeto. Isso é importante, já que suporta o conceito de classificação. Por exemplo, uma deliciosa maça vermelha é parte da classificação ***maçã***, que por sua vez, faz parte da classificação ***frutas***, que está na grande classe ***comida***. Sem o uso de classificações, cada objeto precisaria definir explicitamente todas as suas características. Portanto, usando-se classificações, um objeto precisa definir somente aquelas qualidades que o tornam único dentro de sua classe. Ele pode herdar as qualidades que compartilha com a classe mais geral. É o mecanismo de herança que torna possível a um objeto ser uma instância específica de uma classe mais geral.

2.2 CONSIDERAÇÕES FUNDAMENTAIS SOBRE O C++

O C++ é um superconjunto da linguagem C; portanto, muitos programas em C também são implicitamente programas em C++. Isso significa que é possível escrever programas em C++ idênticos a programas em C. Entretanto, esse processo não é recomendado, pois não extrai todas as vantagens e capacidades do C++.

Ainda que o C++ permita escrever programas idênticos ao C, muitos programadores em C++ usam um estilo e certas características que são exclusivas do C++. Já que é importante aprender a escrever programas em C++ que se pareçam com programas em C++, estão seção introduz um pouco dessas características antes de entrar no “âmago” do C++.

Veja o exemplo a seguir.

```
#include <iostream.h> //biblioteca para operações de E/S em C++

main (void){
```

```
   int i;
   char str[80];

   cout << "C++ é fácil\n";
   printf ("Você pode usar printf() se preferir");

   //para ler um número use
   cout << "Informe um número: ";
   cin >> i;

   //para exibir um número use
   cout << "O seu número é: " << i << "\n";

   //para ler uma string use
   cout << "Informe uma string: ";
   cin >> str;

   //para imprimir uma string use
   cout << str;

   return (0);
}
```

O comando ***cout <<*** é utilizado para enviar dados para um dispositivo de saída, no caso a tela; você também poderia usar ***printf( )***, entretanto ***cout <<*** está mais de acordo com o C++.

O comando ***cin >>*** é utilizado para realizar uma entrada de valores via teclado. Em geral, você pode usar cin >> para carregar uma variável de qualquer um dos tipos básicos mais as strings. Você também poderia usar ***scanf( )*** ao invés de ***cin >>***. Entretanto, cin >> está mais de acordo com o C++.

2.3 COMPILANDO UM PROGRAMA C++

Os compiladores normalmente compilam tanto C como C++. Em geral, se um programa termina em ***.CPP***, ele será compilado como programa C++. Se ele termina com qualquer outras extensão, será compilado como um programa C. Portanto, a maneira mais simples de compilar seus programas C++ como tal é dar a eles a extensão .CPP.

2.4 INTRODUÇÃO A CLASSES E OBJETOS

Conhecidas algumas das convenções e características especiais do C++, é o momento de introduzir a sua característica mais importante: a ***class***. Em C++, para se criar um objeto, deve-se, primeiro, definir a sua forma geral usando-se a palavra reservada ***class***. Uma ***class*** é similar a uma estrutura. Vamos começar com um exemplo. Esta ***class*** define um tipo chamado ***fila***, que é usado para criar um objeto fila:

```
#include <iostream.h>

//isto cria a classe fila
class fila{
   int q[100];
   int sloc, rloc;
public:
   void init(void);
   void qput(int i);
   int qget(void);
};
```

Vamos examinar cuidadosamente esta declaração de ***class***.

Uma ***class*** pode conter tanto partes privadas como públicas. Por definição, todos os itens definidos numa ***class*** são privados. Por exemplo, as variáveis ***q***, ***sloc*** e ***rloc*** são privadas. Isso significa que não podem ser acessadas por qualquer função que não seja membro da ***class***. Essa é uma maneira de se conseguir a encapsulação – a acesso a certos itens de dados pode ser firmemente controlado mantendo-os privados. Ainda que não seja mostrado nesse exemplo, também é possível definir funções privadas que só podem ser chamadas por outros membros de ***class***.

Para tornar partes de uma ***class*** públicas, isto é, acessíveis a outras partes do seu programa, você deve declará-las após a palavra reservada ***public***. Todas as variáveis ou funções definidas depois de ***public*** estão disponíveis a todas as outras funções no programa. Essencialmente, o resto do seu programa acessa um objeto por meio das suas funções e dados ***public***. É preciso ser mencionado que embora se possa ter variáveis ***public***, filosoficamente deve-se tentar limitar ou eliminar seu uso. O ideal é tornar todos os dados privados e controlar o acesso a eles com funções ***public***.

As funções ***init( )***, ***qput( )*** e ***qget( )*** são chamadas de ***funções membros***, uma vez que fazem parte da ***class fila***. Lembre-se de que um objeto forma um laço entre código e dados. Somente aquelas funções declaradas na ***class*** têm acesso às partes privadas da ***class***.

Uma vez definida uma ***class***, pode-se criar um objeto daquele tipo usando-se o nome da ***class***. Essencialmente, o nome da ***class*** torna-se um novo especificador de tipo de dado. Por exemplo, a linha abaixo cria um objeto chamado ***intfila*** do tipo ***fila***:

```
fila intfila;
```

Você também pode criar objetos, quando define a ***class***, colocando os seus nomes após o fechamento das chaves, exatamente como faz com estruturas.

Para revisar, em C++ uma ***class*** cria um novo tipo de dado que pode ser usado para criar objetos daquele tipo.

A forma geral de uma declaração ***class*** é:

```
class nome_de_classe{
   dados e funções privados
```

```
public:
   dados e funções públicos
} lista de objetos;
```

Obviamente, a lista de objetos pode estar vazia.

Dentro da declaração de ***fila***, foram usados protótipos para as funções membros. É importante entender que, em C++, quando é necessário informar ao compilador a respeito de uma função, deve-se usar a forma completa do seu protótipo.

Quando chega o momento de codificar uma função que é membro de uma ***class***, deve ser dito ao compilador a qual ***class*** a função pertence, qualificando-se o nome da função com o nome da ***class*** onde ela é membro. Por exemplo, aqui está uma maneira de codificar a função ***qput( )***:

```
void fila::qput(int i){
   if (sloc == 100){
      cout << "A fila está cheia";
      return;
   }
   sloc++;
   q[sloc] = i;
}
```

O ***::*** é chamado de ***operador de escopo de resolução***. Ele informa ao compilador que essa versão da função ***qput( )*** pertence a ***class fila*** ou, em outras palavras, que a função ***qput( )*** está no escopo de ***fila***. Em C++, muitas ***classes*** diferentes podem usar os mesmos nomes de função. O compilador sabe qual função pertence a qual classe por causa do operador de escopo de resolução e do nome da ***class***.

Para chamar uma função membro de uma parte do seu programa que não faz parte da ***class***, você deve usar o nome do objeto e o operador ponto. Por exemplo, a declaração abaixo chama a função ***init( )*** por meio do objeto ***a***:

```
fila a, b;
a.init();
```

É muito importante entender que ***a*** e ***b*** são dois objetos separados. Isso significa, por exemplo, que a inicialização de ***a*** não faz com que ***b*** também seja inicializado. O único relacionamento entre ***a*** e ***b*** é serem objetos do mesmo tipo.

Outro ponto interessante é que uma função membro pode chamar outra função membro diretamente sem usar o operador ponto. Somente quando uma função membro é chamada por um código que não faz parte da ***class*** é que o operador ponto deve ser usado.

O programa mostrado aqui coloca junto todas as partes e detalhes esquecidos e ilustra a ***class fila***:

```
#include <iostream.h>
```

```
//isto cria a classe fila
class fila{
   int q[100];
   int sloc, rloc;
public:
   void init(void);
   void qput(int i);
   int qget(void);
};

void fila::init(void){
   rloc = sloc = 0;
}

void fila::qput(int i){
   if (sloc == 100){
      cout << "A fila está cheia";
      return;
   }
   sloc++;
   q[sloc] = i;
}

int fila::qget(void){
   if (rloc == sloc){
      cout << "A fila está vazia";
      return (0);
   }
   rloc++;
   return (q[rloc]);
}

main(void){
   fila a, b;   //cria dois objetos fila

   a.init();
   b.init();

   a.qput(10);
   b.qput(19);

   a.qput(20);
   b.qput(1);
   cout << a.qget() << " ";
   cout << a.qget() << " ";
   cout << b.qget() << " ";
   cout << b.qget() << " ";
```

```
   return (0);
}
```

LEMBRE-SE: As partes privadas de um objeto são acessíveis somente às funções membros daquele objeto. Por exemplo, uma declaração como:

```
a.rloc = 0;
```

não poderia estar na função ***main( )*** do programa anterior.

2.5 SOBRECARGA DE FUNÇÕES

Uma maneira do C++ obter polimorfismo é pelo uso de sobrecarga de funções. Em C++ duas ou mais funções podem compartilhar o mesmo nome, contanto que as suas declarações de parâmetros sejam diferentes. Nessa situação, as funções que compartilham o mesmo nome são conhecidas como ***sobrecarregadas*** e o processo é chamado de ***sobrecarga de funções***. Por exemplo, considere este programa:

```
#include <iostream.h>
//a função quadrado é sobrecarregada três vezes
int quadrado (int i);
double quadrado (double d);
long quadrado (long l);

main(void){
   cout << quadrado(10) << "\n";
   cout << quadrado(11.0) << "\n";
   cout << quadrado(9L) << "\n";
   return(0);
}

int quadrado(int i){
   cout << "Dentro da função quadrado() que usa ";
   cout << "um argumento inteiro.\n";
   return (i * i);
}

double quadrado(double d){
   cout << "Dentro da função quadrado() que usa ";
   cout << "um argumento double.\n";
   return (d * d);
}

long quadrado(long l){
   cout << "Dentro da função quadrado() que usa ";
   cout << "um argumento long.\n";
```

```
   return (l * l);
}
```

Esse programa cria três funções similares, porém diferentes, chamadas de ***quadrado( )***. Cada uma delas retorna o quadrado do seu argumento. Como o programa ilustra, o compilador sabe qual função usar em cada situação por causa do tipo de argumento.

O mérito da sobrecarga de funções é permitir que conjuntos relacionados de funções sejam acessados usando-se somente um nome. Nesse sentido, a sobrecarga de função deixa que se crie um nome genérico para algumas operações, com o compilador resolvendo exatamente qual função é necessária no momento para realizar a operação.

O que torna a sobrecarga de funções importante é o fato de ela poder ajudar no tratamento de complexidades. Para entender como, considere este exemplo. Muitos compiladores de linguagem C contêm funções como ***atoi( )***, ***atof( )*** e ***atol( )*** nas suas bibliotecas-padrões. Coletivamente, essas funções convertem uma string de dígitos em formatos internos de inteiros, double e long, respectivamente. Embora essas funções realizem ações quase idênticas, três nomes completamente diferentes devem ser usados em C para representar essas tarefas, o que torna a situação mais complexa do que é na realidade.

Ainda que o conceito fundamental de cada função seja o mesmo, o programador tem três coisas para se lembrar, e não somente uma. Entretanto, em C++, é possível usar o mesmo nome, ***atonum( )***, por exemplo, para todas as três funções. Assim, o nome ***atonum( )*** representa a ***ação geral*** que está sendo realizada. É de responsabilidade do compilador selecionar a versão ***específica*** para uma circunstância particular. Assim, o programador só precisa lembrar da ação geral que é realizada. Portanto, aplicando-se o polimorfismo, três coisas a serem lembradas foram reduzidas a uma. Ainda que esse exemplo seja bastante trivial, se você expandir o conceito, verá como o polimorfismo pode ajudar na compreensão de programas muito complexos.

Um exemplo mais prático de sobrecarga de funções é ilustrado pelo programa seguinte. Como você sabe, a linguagem C (e o C++) não contém nenhuma função de biblioteca que solicite ao usuário uma entrada, esperando, uma resposta. Este programa cria três funções, chamadas ***solicitacao( )***, que realizam essa tarefa para dados dos tipos ***int***, ***double*** e ***long***:

```
#include <iostream.h>

void solicitacao (char *str, int *i);
void solicitacao (char *str, double *d);
void solicitacao (char *str, long *l);

main(void){
   int i;
   double d;
   long l;

   solicitacao("Informe um inteiro: ", &i);
   solicitacao("Informe um double: ", &d);
   solicitacao("Informe um long: ", &l);
```

```
   return(0);
}

void solicitacao (char *str, int *i){
   cout << str;
   cin >> *i;
}

void solicitacao (char *str, double *d){
   cout << str;
   cin >> *d;
}

void solicitacao (char *str, long *l){
   cout << str;
   cin >> *l;
}
```

CUIDADO: Você pode usar o mesmo nome para sobrecarregar funções não relacionadas, mas não deve faze-lo. Por exemplo, você pode usar o nome ***quadrado( )*** para criar funções que retornam o quadrado de um ***int*** e a raiz quadrada de um ***double***. Entretanto, essas duas operações são fundamentalmente diferentes e a aplicação de sobrecarga de função, dessa maneira, desvirtua inteiramente o seu propósito principal. Na prática, você somente deve usar sobrecarga em operações intimamente relacionadas.

2.6 SOBRECARGA DE OPERADOR

Uma outra maneira de se obter polimorfismo em C++ é pela sobrecarga de operador. Como você sabe, em C++m pode-se usar os operadores << e >> para realizar operações de E/S em console. Isso é possível porque, no arquivo ***iostream.h***, esses operadores são sobrecarregados, tomando, assim, um significado adicional relativo a uma certa ***class***. Entretanto, ele ainda mantém o seu antigo significado (deslocamento de bits).

Em geral, você pode sobrecarregar qualquer um dos operadores do C++ definindo qual significado eles têm em relação a uma ***class*** específica. Por exemplo, recordando a ***class fila*** desenvolvida anteriormente neste capítulo, é possível sobrecarregar o operador **+** relativo aos objetos do tipo ***fila*** de maneira que ele acrescente o conteúdo de uma pilha em uma outra já existente. Contudo, o operador **+** ainda retém seu significado original em relação a outros tipos de dados. Mais a frente será apresentado um exemplo de sobrecarga de operador.

2.7 HERANÇA

Como definido, a herança é uma das principais características de uma linguagem de programação orientada a objetos. Em C++, a herança é suportada por permitir a uma ***class*** incorporar outra ***class*** na sua declaração. Para ver como isso

funciona, vamos começar com um exemplo. Aqui está uma ***class***, chamada ***veiculos_na_estrada***, que define claramente veículos que trafegam nas estradas. Ela armazena o número de rodas que um veículo tem e o número de passageiros que pode transportar:

```
class veiculos_na_estrada{
   int rodas;
   int passageiros;
public:
   void fixa_rodas(int num);
   int obtem_rodas(void);
   void fixa_passageiros(int num);
   int obtem_passageiros(void);
};
```

Essa rude definição de uma estrada de rodagem pode ser usada para ajudar a determinar objetos específicos. Por exemplo, este código declara uma ***class*** chamada ***caminhão*** usando ***veiculos_na_estrada***:

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

Note como ***veiculos_na_estrada*** é herdada. A forma geral para herança é mostrada aqui:

```
class nome_da_nova_classe : acesso classe_herdada{...
```

Aqui, ***acesso*** é opcional. Contudo, se estiver presente, ele deve ser ***public***, ***private*** ou ***protected***. A utilização de ***public*** significa que todos os elementos ***public*** do antepassado também serão ***public*** para a ***class*** que os herda.

Portanto, os membros da ***class caminhao*** têm acesso às funções membros de ***veiculos_na_estrada***, exatamente como se tivessem sido declarados dentro da ***class caminhao***. Entretanto, as funções membros ***não*** têm acesso às partes privadas da ***class veiculos_na_estrada***. Aqui está um programa que ilustra herança. Ele cria duas subclasses ***veiculos_na_estrada*** usando herança. Uma é ***caminhao***, a outra, ***automovel***.

```
#include <iostream.h>

class veiculos_na_estrada{
   int rodas;
   int passageiros;
public:
```

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

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

enum tipo {carro, furgão, caminhão};

class automovel : public veiculos_na_estrada{
   enum tipo tipo_de_carro;
public:
   void fixa_tipo(enum tipo t);
   enum tipo obtem_tipo(void);
   void exibe(void);
};

void veiculos_na_estrada::fixa_rodas(int num){
   rodas = num;
}

int veiculos_na_estrada::obtem_rodas(void){
   return rodas;
}

void veiculos_na_estrada::fixa_passageiros(int num){
   passageiros = num;
}

int veiculos_na_estrada::obtem_passageiros(void){
   return passageiros;
}

void caminhao::fixa_carga(int num){
   carga = num;
}

int caminhao::obtem_carga(void){
   return carga;
}
```

```
void caminhao::exibe(void){
   cout << "rodas: " << obtem_rodas() << "\n";
   cout << "passageiros: " << obtem_passageiros() << "\n";
   cout << "capacidade de carga em metros cúbicos: " <<
        obtem_rodas() << "\n";
}

void automovel::fixa_tipo(enum tipo t){
   tipo_de_carro = t;
}

enum tipo automovel::obtem_tipo(void){
   return tipo_de_carro;
}

void automovel::exibe(void){
   cout << "rodas: " << obtem_rodas() << "\n";
   cout << "passageiros: " << obtem_passageiros() << "\n";
   cout << "tipo: ";

   switch(obtem_tipo(){
      case furgão:   cout << "furgão\n";
                     break;
      case carro:    cout << "carro\n";
                     break;
      case caminhão: cout << "caminhão\n";
   }
}

main(void){
   caminhao t1, t2;
   automovel c;

   t1.fixa_rodas(18);
   t1.fixa_passageiros(2);
   t1.fixa_carga(3200);

   t1.fixa_rodas(6);
   t1.fixa_passageiros(3);
   t1.fixa_carga(1200);

   t1.exibe();
   t2.exibe();

   c.fixa_rodas(4);
   c.fixa_passageiros(6);
   c.fixa_tipo(furgão);
```

```
   c.exibe();

   return(0);
}
```

Como esse programa mostra, a maior vantagem da herança é poder criar uma classificação base possível de ser incorporada por outras mais específicas. Dessa maneira, cada objeto pode representar precisamente a sua própria classificação.

Note que, tanto ***caminhao*** como ***automovel***, incluem a função membro chamada ***exibe( )***, que mostra informações sobre cada objeto. Esse é um outro aspecto do polimorfismo. Já que cada função ***exibe( )*** está relacionada à sua própria classe, o compilador pode facilmente dizer qual chamar em qualquer circunstância.

2.8 CONSTRUTORES E DESTRUTORES

É muito comum alguma parte de um objeto requerer inicialização antes de ser usada. Por exemplo, voltando à ***class fila*** desenvolvida neste capítulo, antes da fila poder ser usada, as variáveis ***rloc*** e ***sloc*** são fixadas em zero. Isso foi realizado por meio da função ***init( )***. Já que a exigência de inicialização é tão comum, o C++ permite aos objetos serem inicializados por si mesmos quando são criados. Essa inicialização automáticas é realizada pelo uso de uma função ***construtora***.

A construtora é uma função especial que é membro da ***class*** e tem o mesmo nome que a ***class***. Por exemplo, aqui está com a ***class fila*** fica quando convertida para usar uma função construtora para inicialização.

```
//O código seguinte cria a classe fila
class fila{
   int q[100];
   int sloc, rloc;
public:
   fila(void);       //construtor
   void qput(int i);
   int qget(void);
};
```

Note que o construtor ***fila( )*** não possui tipo de retorno especificado. Em C++, funções construtoras não podem retornar valores.

O construtor ***fila( )*** é implementado assim:

```
//Este é o construtor
fila::fila(void){
   sloc = rloc = 0;
   cout << "fila inicializada\n";
}
```

Tenha em mente que a mensagem ***fila inicializada*** é emitida como uma maneira de ilustrar o construtor. Na prática, muitas funções construtoras não emitirão

nem receberão qualquer coisa.

Um construtor de objeto é chamado quando o objeto é criado. Isso significa que ele é chamado quando a declaração do objeto é executada. Ainda, para objetos locais, o construtor é chamado toda vez que a declaração do objeto é encontrada.

O complemento do construtor é o ***destrutor***. Em muitas circunstâncias, um objeto precisará realizar alguma ação quando é destruído. Por exemplo, um objeto pode precisar desalocar memória previamente alocada. Em C++, é a função destrutora que manipula desativações. O destrutor tem o mesmo nome que o construtor, mas é precedido por um **~**. Por exemplo, aqui está a ***class fila*** e as suas funções construtora e destrutora. Lembre-se que a classe ***fila*** não requer um destrutor, de modo que o destrutor mostrado aqui é somente para ilustração.

```
#include <iostream.h>

//o código seguinte cria a classe fila
class fila{
   int q[100];
   int sloc, rloc;
public:
   fila(void);   //construtor
   ~fila(void);  //destrutor
   void qput(int i);
   int qget(void);
};

//Esta é a função construtora
fila::fila(void){
   sloc = rloc = 0;
   cout << "Fila inicializada\n";
}

//Esta é a função destrutora
fila::~fila(void){
   cout << "Fila destruída\n";
}

void fila::qput(int i){
   if (sloc == 100){
      cout << "A fila está cheia";
      return;
   }
   sloc++;
   q[sloc] = i;
}

int fila::qget(void){
   if (rloc == sloc){
```

```
      cout << "A fila está vazia";
      return (0);
   }
   rloc++;
   return (q[rloc]);
}

main(void){
   fila a, b;   //cria dois objetos fila

   a.qput(10);
   b.qput(19);
   a.qput(20);
   b.qput(1);

   cout << a.qget() << " ";
   cout << a.qget() << " ";
   cout << b.qget() << " ";
   cout << b.qget() << " ";
   return (0);
}
```

Esse programa exibe o seguinte:

```
fila inicializada
fila inicializada
10 20 19 1
fila destruída
fila destruída
```

2.9 FUNÇÕES FRIEND

É possível para uma função não-membro de uma ***class*** ter acesso a partes privadas de uma ***class*** pela declaração da função como uma ***friend*** da ***class***. Por exemplo, aqui, a função ***frd( )*** é declarada para ser uma função ***friend*** da ***class c1***:

```
class c1{
⋮
public:
   friend void frd(void);
⋮
};
```

Como você pode ver, a palavra reservada ***friend*** precede a declaração inteira da função, que é o caso geral.

Ainda que não tecnicamente necessárias, as funções ***friend*** são permitidas em C++ para acomodar uma situação em que, por questões de eficiência, duas classes devam compartilhar a mesma função. Para ver um exemplo, considere um programa que define duas classes chamadas ***linha*** e ***box***. A ***class linha*** contém todos os dados e código necessários para desenhar uma linha horizontal tracejada de qualquer tamanho, começando nas coordenadas X, Y determinadas e usando uma cor especificada. A ***class Box*** contém todo código e dados necessários para desenhar um Box nas coordenadas especificadas pelo canto superior esquerdo e canto inferior direito em uma cor determinada. As duas ***classes*** usam a mesma cor. Essas ***classes*** são declaradas como mostrado aqui:

```
class linha;

class box{
   int cor;          //cor do box
   int upx, upy;     //canto superior esquerdo
   int lowx, lowy.   //canto inferior direito
public:
   friend int mesma_cor(linha l, box b)
   void indica_cor(int c);
   void define_box(int x1, int y1, int x2, int y2);
   void exibe_box(void);
};

class linha{
   int cor;
   int comecox, comecoy;
   int tamanho;
public:
   friend int mesma_cor(linha l, box b);
   void indica_cor(int c);
   void define_linha(int x, int y, int l);
   void exibe_linha();
};
```

A função ***mesma_cor( )***, que não é membro de nenhuma mas, sim, ***friend*** das duas, retorna verdadeiro se tanto o objeto ***linha*** como o objeto ***box***, que formam os seus argumentos, são desenhados na mesma cor; retorna zero, caso contrário. A função ***mesma_cor( )*** é mostrada aqui:

```
//retorna verdadeiro se a linha e o box têm a mesma cor
int mesma_cor(linha l, box b){
   if (l.cor == b.cor) return (1);
   return (0);
}
```

Como você pode ver, a função ***mesma_cor( )*** precisa ter acesso a partes privadas, tanto de ***linha*** como de ***box***, para realizar sua tarefa eficientemente.

Note a declaração vazia de ***linha*** no começo das declarações ***class***. Já que a função ***mesma_cor( )*** em ***box*** referencia ***linha*** antes que ela seja declarada, ***linha*** deve ser referenciada de antemão. Se isso não for feito, o compilador não saberá o que ela é quando encontrá-la na declaração de ***box***. Em C++, uma referência antecipada a uma classe é simplesmente uma palavra reservada ***class*** seguida pelo nome da ***class***. Usualmente, as referências antecipadas só são necessárias quando as funções ***friend*** estão envolvidas.

Aqui está um programa que demonstra as classes ***linha*** e ***box*** e ilustra como uma função ***friend*** pode acessar partes privadas de uma ***class***. Este programa faz uso de várias funções de tela do compilador Borland C++.

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

class linha;

class box{
   int cor;          //cor do box
   int upx, upy;     //canto superior esquerdo
   int lowx, lowy;   //canto inferior direito
public:
   friend int mesma_cor(linha l, box b)
   void indica_cor(int c);
   void define_box(int x1, int y1, int x2, int y2);
   void exibe_box(void);
};

class linha{
   int cor;
   int comecox, comecoy;
   int tamanho;
public:
   friend int mesma_cor(linha l, box b);
   void indica_cor(int c);
   void define_linha(int x, int y, int l);
   void exibe_linha();
};

//retorna verdadeiro se a linha e o box têm a mesma cor
int mesma_cor(linha l, box b){
   if (l.cor == b.cor) return (1);
   return (0);
}

void box::indica_cor(int c){
   cor = c;
}
```

```
void linha::indica_cor(int c){
   cor = c;
}

void box::define_box(int x1, int y1, int x2, int y2){
   upx = x1;
   upy = y1;
   lowx = x2;
   lowy = y2;
};

void box::exibe_box(void){
   int i;

   textcolor(cor);
   gotoxy(upx, upy);
   for (i = upx; i <= lowx; i++) cprintf("-");
   gotoxy (upx, lowy - 1);
   for (i = upx; i <= lowx; i++) cprintf("-");

   gotoxy(upx, upy);
   for (i = upy; i <= lowy; i++){
      cprintf("|");
      gotoxy(upx, i);
   }

   gotoxy(lowx, upy);
   for (i = upy; i <= lowy; i++){
      cprintf("|");
      gotoxy(lowx, i);
   }
}

void linha::define_linha(int x, int y, int l){
   comecox = x;
   comecoy = y;
   tamanho = l;
}

void linha::exibe_linha(void){
   int i;

   textcolor(cor);
   for (i = 0; i < tamanho; i++) cprintf("-");
}

main(void){
```

```
   box b;
   linha l;
   b.define_box(10, 10, 15, 15);
   b.indica_cor(3);
   b.mostra_box();
   l.define_linha(2, 2, 10);
   l.indica_cor(2);
   l.exibe_linha();

   if (!mesma_cor(l, b)) cout << "Não são da mesma cor";

   cout << "\nPressione qualquer tecla";
   getche();

   //agora, torne a linha e o box da mesma cor
   l.define_linha(2, 2, 10);
   l.indica_cor(3);
   l.exibe_linha();

   if (mesma_cor(l, b)) cout << "São da mesma cor";
   return(0);
}
```

2.10 A PALAVRA RESERVADA this

Para melhor compreender a sobrecarga de operadores, é necessário aprender a respeito de outra palavra reservada do C++, chamada ***this***, que é um ingrediente essencial para muitas sobrecargas de operadores.

Cada vez que se invoca uma função membro, automaticamente é passado um ponteiro para o objeto que a invoca. Você pode acessar esse ponteiro usando a palavra reservada ***this***. O ponteiro ***this*** é um parâmetro ***implícito*** para todas as funções membros.

Uma função membro pode acessar diretamente os dados ***private*** da sua ***class***. Por exemplo, dado esta ***class***:

```
class cl{
   int i;
⋮
};
```

uma função membro pode atribuir a ***i*** o valor 10 usando esta declaração:

```
i = 10;
```

A declaração anterior é uma abreviação para esta declaração:

```
this->i = 10;
```

Para ver como o ponteiro ***this*** funciona, examine o pequeno programa seguinte:

```
#include <iostream.h>
class cl{
   int I;
public:
   void carrega_i(int val){
      this->i = val;     //o mesmo que i = val
   }
   int obtem_i(void){
      return(this->i);   //o mesmo que return i
   }
};

void main(void){
   cl o;

   o.carrega_i(100);
   cout << o.obtem_i();
}
```

Esse programa exibe o número 100.

Ainda que o exemplo anterior seja trivial – de fato, ninguém deve usar o ponteiro ***this*** dessa maneira –, na próxima seção você verá porque o ponteiro ***this*** é tão importante.

2.11 SOBRECARGA DE OPERADOR – MAIORES DETALHES

Outra característica do C++ relacionada com sobrecarga de funções é a chamada ***sobrecarga de operadores***. Com pouquíssimas exceções, muitos dos operadores do C++ podem receber significados especiais relativos a classes específicas. Por exemplo, uma ***class*** que define uma lista ligada pode usar o operador **+** para adicionar um objeto à lista. Outras ***class*** pode usar o operador **+** de uma maneira inteiramente diferente. Quando um operador é sobrecarregado, nada do seu significado original é perdido; simplesmente é definida uma nova operação relativa a uma ***class*** específica. Portanto, sobrecarregar o operador **+** para manipular uma lista ligada não faz com que, por exemplo, o seu significado relativo aos inteiros (isto é, adição) seja mudado.

Para sobrecarregar um operador, você deve definir o que a dada operação significa em relação à ***class*** em que ela é aplicada. Para isso, crie uma função ***operator*** que defina sua ação. A forma geral da função ***operator*** é mostra aqui:

```
tipo nome_da_classe::operator#(lista de argumentos){
   //operação definida relativa à classe
}
```

Aqui, ***tipo*** é o tipo do valor retornado pela operação especificada. Freqüentemente, o valor de retorno é do mesmo tipo que a ***class*** (ainda que possa ser de qualquer tipo que você selecionar). Um operador sobrecarregado tem freqüentemente um valor de retorno do mesmo tipo de ***class*** para onde é sobrecarregado porque facilita o seu uso em expressões complexas.

Funções operator devem ser ou membros ou ***friends*** da ***class*** na qual estão sendo usadas. Para ver como a sobrecarga de operadores funciona, vamos começar com um exemplo simples que cria uma ***class***, chamada ***tres_d***, que mantém as coordenadas de um objeto no espaço tridimensional. Este programa sobrecarrega os operadores **+** e **=** relativos à ***class tres_d***. Examine-o cuidadosamente:

```
#include <iostream.h>

class tres_d{
   int x, y, z;      //coordenadas 3-D
public:
   tres_d operator+ (tres_d t);
   tres_d operator= (tres_s t);
   void mostra(void);
   void atribui(int mx, int my, int mz);
};

//sobrecarrega o +
tres_d tres_d::operator+ (tres_d t){
   tres_d temp;

   temp.x = x + t.x;
   temp.y = y + t.y;
   temp.z = z + t.z;
   return(temp);
}

//sobrecarrega o =
tres_d tres_d::operator= (tres_d t){
   x = t.x;
   y = t.y;
   z = t.z;
   return(*this);
}

//mostra as coordenadas x, y, z
tres_d tres_d::mostra(void){
   cout << x << ", ";
   cout << y << ", ";
   cout << z << "\n";
}
```

```
//atribui coordenadas
void tres_d::atribui(int mx, int my, int mz){
   x = mx;
   y = my;
   z = mz;
}

main(void){
   tres_d a, b, c;

   a.atribui(1, 2, 3);
   b.atribui(10, 10, 10);
   a.mostra();
   b.mostra();
   c = a + b;       //agora adiciona a e b
   c.mostra();
   c = a + b + c;  //agora adiciona a, b e c
   c.mostra();
   c = b = a;       //demonstra atribuição múltipla
   c.mostra();
   b.mostra();
   return(0);
}
```

Esse programa produz o seguinte:

```
1, 2, 3
10, 10, 10
11, 12, 13
22, 24, 26
1, 2, 3
1, 2, 3
```

Note que as duas funções operator têm somente um parâmetro cada, embora sobrecarreguem operações binárias. A razão para essa aparente contradição é que, quando um operador binário é sobrecarregado usando-se uma função membro, somente um argumento precisa ser passado para ele explicitamente. O outro argumento é passado implicitamente usando o ponteiro ***this***. Assim, na linha

```
temp.x = x + t.x;
```

o ***x*** refere-se a ***this.x***, que é o ***x*** associado ao objeto que solicitou a chamada para a função operator. Em todos os casos, é o objeto no lado esquerdo de uma operação que faz com que seja chamada a função operator. O objeto no lado direito é passado para a função.

Em geral, quando você está sobrecarregando uma função membro,

nenhum parâmetro é necessário para sobrecarregar um operador unário e somente um é requerido para sobrecarregar um operador binário. Em ambos os casos, o objeto que causa a ativação da função operator é implicitamente passado pelo ponteiro ***this***.

Para entender como a sobrecarga de operador funciona, vamos examinar esse programa cuidadosamente, começando como o operador sobrecarregado **+**. Quando dois objetos do tipo ***tres_d*** são operados pelo operador **+**, a amplitude das suas respectivas coordenadas são adicionadas, como mostrado na função ***operator+( )*** associada a essa ***class***. Note, entretanto, que essa função não modifica o valor de qualquer operando. Em vez disso, um objeto do tipo ***tres_d*** é retornado pela função que contém o resultado da operação. Esse é um ponto importante.

Para entender por que a operação **+** não deve mudar o conteúdo de qualquer objeto, pense a respeito da operação aritmética padrão ***10 + 12***. O resultado dessa operação é 22, porém nem 10 nem 12 são modificados pela operação.

Outro ponto-chave sobre como o operador **+** é sobrecarregado é que ele retorna um objeto do tipo ***tres_d***. Apesar da função poder retornar qualquer tipo válido do C++, o fato de ela retornar um objeto ***tres_d*** permite ao operador **+** ser usado em expressões mais complexas, tal como ***a + b + c***.

Contrastando com o **+**, o operador de atribuição, na verdade, faz com que um dos seus argumentos seja modificado. Já que a função ***operator=( )*** é chamada pelo objeto que ocorre no lado esquerdo da atribuição, é ele que é modificado pela operação de atribuição. Entretanto, até mesmo a operação de atribuição deve retornar um valor, haja visto que, em C++, a operação de atribuição produz um valor que ocorre no lado direito. Assim, para permitir uma declaração como:

```
a = b = c = d;
```

é necessário que a função ***operator+( )*** retorne o objeto apontado por ***this***, que será aquele que ocorre no lado esquerdo da declaração de atribuição. Isso permite que uma cadeia de atribuições seja feita.

Também é possível sobrecarregar operadores unários, como **++** e **--**. Como definido anteriormente, quando se sobrecarrega um operador unário, nenhum objeto é explicitamente passado para a função operator. Em vez disso, a operação é realizada no objeto que gera a chamada para a função pela passagem implícita do ponteiro ***this***. Por exemplo, aqui está uma versão expandida do exemplo anterior, que define as operações de incremento para os objetos do tipo ***tres_d***:

```
#include <iostream.h>

class tres_d{
   int x, y, z;      //coordenadas 3-D
public:
   tres_d operator+ (tres_d op2);  //op1 está implícito
   tres_d operator= (tres_d op2);  //op1 está implícito
   tres_d operator++ (void);       //op1 também está implícito
   void mostra(void);
   void atribui(int mx, int my, int mz);
};
```

```
tres_d tres_d::operator+ (tres_d op2){
   tres_d temp;

   temp.x = x + op2.x;   //estas são adições de inteiros
   temp.y = y + op2.y;   //e o + retém o seu significado
   temp.z = z + op2.z;   //original relativo a eles
   return(temp);
}

tres_d tres_d::operator= (tres_d op2){
   x = op2.x;            //estas são atribuições de inteiros
   y = op2.y;            //e o = retém o seu significado
   z = op2.z;            //original relativo a eles
   return(*this);
}

//sobrecarrega um operador unário
tres_d tres_d::operator++ (void){
   x++;
   y++;
   z++;
   return(*this);
}

//mostra as coordenadas x, y, z
void tres_d::mostra(void){
   cout << x << ", ";
   cout << y << ", ";
   cout << z << "\n";
}

//atribui coordenadas
void tres_d::atribui(int mx, int my, int mz){
   x = mx;
   y = my;
   z = mz;
}

int main( ){
   tres_d a, b, c;

   a.atribui(1, 2, 3);
   b.atribui(10, 10, 10);
   a.mostra();
   b.mostra();
   c = a + b;      //agora adiciona a e b
```

```
   c.mostra();
   c = a + b + c;  //agora adiciona a, b e c
   c.mostra();
   c = b = a;      //demonstra atribuição múltipla
   c.mostra();
   b.mostra();
   ++c;            //incrementa c
   c.mostra();
   return(0);
}
```

Um ponto importante quando se sobrecarrega os operadores **++** ou **–** é que não é possível determinar de dentro da função ***operator*** se o operador precede ou segue seu operando. Isto é, a sua função ***operator*** não pode saber se a expressão que causa a chamada para a função é:

```
++OBJ;
```

ou

```
OBJ++;
```

onde ***OBJ*** é o objeto da operação.

A ação de um operador sobrecarregado, como é aplicada à ***class*** para a qual é definido, não precisa sustentar qualquer relacionamento com o uso do operador-padrão quando aplicado aos tipos próprios do C++. Por exemplo, **<<** e **>>** quando aplicados a ***cout*** e ***cin***, não têm nada em comum com quando aplicados a tipos inteiros. Contudo, para propósitos de estruturação e legibilidade do seu código, um operador sobrecarregado deve refletir, quando possível, o espírito do uso original do operador. Por exemplo, o operador **+** relativo a ***class tres_d*** é conceitualmente similar ao **+** relativo aos tipos inteiros. Há poucos benefícios, por exemplo, na definição do operador **+** relativo a alguma ***class*** de tal forma que atue de modo completamente inesperado. O conceito-chave aqui é que, enquanto se pode dar a um operador sobrecarregado qualquer significado que se deseja, é melhor, para efeito de esclarecimento, que o novo significado esteja relacionado com o original.

Existem algumas restrições aplicáveis aos operadores sobrecarregados. Primeiro, não se pode alterar a precedência de qualquer operador. Segundo, não se pode alterar o número de operandos requeridos pelo operador, ainda que a função ***operator*** possa optar por ignorar um operando. Finalmente, com exceção do operador **=**, os operadores sobrecarregados são herdados por qualquer ***class*** derivada. Cada ***class*** deve definir explicitamente o seu próprio operador **=** sobrecarregado, se for necessário.

Os únicos operadores que não podem ser sobrecarregados são mostrados aqui: “***.***”, “***::***”, “***.****”, “***?***”.