

Capítulo 3

Encapsulamento

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

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

Controle de acesso - public e private

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

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

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

O exemplo abaixo ilustra o uso destas novas palavras reservadas:

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

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

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

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

```
struct SemUso {  
    private:
```

```

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

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

```

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

```

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

```

Membros *protected* serão explicados quando forem introduzidas as classes derivadas.

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

```

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

```

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

```

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

```

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

Declaração de classes com a palavra reservada class

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

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

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

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

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

Classes e funções friend

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

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

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

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

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

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

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

```
class Conjunto {
    friend class IteradorConj;

    struct listElem {
```

...
...

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

```
class No {  
    friend int leValor( No* ); // dá acesso privilegiado  
                                // à função leValor  
  
    int valor;  
public:  
    void setaValor( int v ) { valor=v; }  
};  
  
int leValor( No* n )  
{  
    return n->valor; // acessa dado private de No  
}
```

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

Exercício 3 - Calculadora RPN com controle de acesso

Modificar a calculadora de acordo com o controle de acesso

Construtores e destrutores

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

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

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

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

Declaração de construtores e destrutores

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

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

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

```
class Conjunto {  
    friend class IteradorConj;  
    struct listElem {  
        listElem *prox;  
        int valor;  
    };  
    listElem* lista;  
    int nElems;
```

```

public:
    Conjunto() { nElems=0; lista=NULL; }
    ~Conjunto(); // desaloca os nós da lista
    void insere(int n);
    void retira(int n);
    int pertence(int n);
    int cardinalidade();
};

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

```

Chamada de construtores e destrutores

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

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

```

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

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

```

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

```

A global;

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

```

Este código produz a seguinte saída:

```

construtor
main
destrutor

```

Construtores com parâmetros

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

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

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

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

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

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

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

Construtores gerados automaticamente

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

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

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

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

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

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

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

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

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

```

    x a3 = a2; // usa construtor de cópia
}

```

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

Objetos temporários

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

```

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

void f(A);

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

```

Conversão por construtores

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

```

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

void f(A);

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

```

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

```

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

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

```

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

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

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

Construtores privados

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

Destrutores privados

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

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

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

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

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

Inicialização de campos de classes com construtores

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

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

A a(123);
```

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

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



```
class B {  
    A a;  
};
```

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

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

Exercício 4 - Calculadora RPN com construtores

Utilizar construtores na calculadora

