UTFPR - UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Campus Campo Mourão

[1]Alexandre Aparecido Scrocaro Junior R.A.: 2135485

[1]alexandre.2001@alunos.utfpr.edu

RESENHA DOS CAPÍTULOS 3 E 5 DO LIVRO "Programação orientada a objetos com c++" de Renato Borges e André Luiz Clinio

Capítulo 3:

1. Encapsulamento:

O TAD é moldado conforme sua utilização, assim pode ser implementado de diversas formas. Os atributos na implementação não devem ser disponibilizados ao usuário de um objeto. O que foi visto até esse ponto do livro não permite o encapsulamento dos dados. Logo, no presente capítulo, formas de fazê-lo são apresentadas.

1.1 Controle de acesso - public e private:

Aqui tem-se as formas de esconder o máximo de informações possível através de restrições na manipulação de classes, dados e códigos. São elas:

PRIVATE	O mais restrito: somente a própria classe pode acessá-los, ou seja, somente os métodos da própria classe tem acesso a eles.
PUBLIC	Estes atributos são públicos, logo são passíveis de serem chamados em outras classes que não sejam a sua.

Tabela 1: Definições de public e private.
Fonte: Adaptada do livro.

Ainda há a protected, porém ela não foi introduzida nesse momento.

1.2 Declaração de classes com a palavra reservada class

A palavra class pode ser usada para substituir struct. A diversidade entre elas é que enquanto os membros de uma struct são públicos, os da class são privados (apenas os explicitamente definidos como public podem ser exportados).

1.3 Classes e funções friend

Quando duas classes são muito próximas ao ponto de precisar interagir. A solução normalmente é usar iteradores (os quais atuam sobre alguma coleção de elementos e a cada chamada retorna um elemento diferente da coleção, até que já tenha retornado todos). Porém, o iterador utiliza dados privados, gerando erros na compilação; Então, como as classes estão intimamente ligadas (como visto anteriormente), seria conveniente que uma classe tivesse acesso irrestrito à outra, para tanto existe a palavra reservada friend. Ela oferece acesso especial às classes ou funções.

Ao declarar funções friend, ela terá acesso irrestrito aos componentes da classe que a declarou como friend. O recurso de classes e funções friend devem ser usados com cuidado, pois isto é um furo no encapsulamento dos dados de uma classe.

2. Construtores e destrutores:

Os nomes já indicam suas funções para uma classe. Eles são chamados automaticamente assim que um objeto é criado ou destruído. Os construtores garantem que um objeto recém criado esteja consistente. O compilador garante que o construtor é a primeira função a ser executada sobre um objeto. Já os destrutores são utilizados para liberar recursos alocados pelo objeto, como memória e arquivos.

2.1 Declaração de construtores e destrutores

Ambos não têm um retorno, e o destrutor não pode receber parâmetros, ao contrário do construtor. A seguir tem-se um exemplo de como eles podem ser utilizados.

```
class Conjunto {
        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);
};
```

2.2 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, como mostra o exemplo:

```
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
```

2.3 Construtores e destrutores privados

Eles podem, assim como qualquer método, ser privados. Os objetos só poderão ser criados ou destruídos quando o construtor ou destrutor estiverem dentro de métodos da própria classe ou em classes e funções friend. fazendo isso, é possível projetar classes cujos objetos não são nunca destruídos.

Capítulo 5:

1. Aspectos de reutilização

É de conhecimento de qualquer programador que o desenvolvimento de software é repetitivo: mesmo padrão de ordenação, busca de um elemento, comparação, etc em programas diferentes. Logo, pode-se concluir que, apesar dos programadores tenderem a escrever os mesmos tipos de rotinas diversas vezes, estas não são exatamente iguais: muitos detalhes mudam de implementação para implementação (tipos dos elementos, estrutura de dados associada, etc.).

Apesar disso, algumas soluções foram propostas com relativos sucessos:

Reutilização de código-fonte	Muito comum no meio científico. Muito da cultura UNIX foi difundida pelos laboratórios e universidades do mundo graças à disponibilidade de código-fonte. Esta técnica não suporta ocultação de informação
Reutilização de pessoal	É uma forma muito comum na indústria. Consiste em transferir equipes de desenvolvedores para assegurar a aplicação de experiências passadas em novos projetos. Obviamente, esta é uma maneira não-técnica e limitada.
Reutilização de design	A ideia por trás desta técnica é que as companhias devem acumular repositórios de idéias descrevendo designs utilizados para os tipos de aplicação mais comuns.

Tabela 2: Soluções para reutilização de códigos
Fonte: livro

1.1 Requisitos para reuso

As ideias supracitadas lembram que: o software é definido pelo seu código, logo, deve-se produzir programas reutilizáveis. A reutilização de pessoal, apesar de limitada, é fundamental pois o desenvolvimento de um software precisa de profissionais bem treinados e com experiência para reconhecer as situações com possibilidade de reuso. A questão do design enfatiza a necessidade de componentes reutilizáveis que estejam em um nível conceitual e de generalidade alto.

A seguir, são expostos conceitos a ser ponderados na possibilidade de reuso:

- Variação no tipo: Um módulo deve ser capaz de funcionar sobre qualquer tipo a ele atribuído.

- Variação nas estruturas de dados e algoritmos: A busca de um elemento pode ser adaptada para diversos tipos de estruturas de dados e algoritmos de busca: tabelas sequenciais, vetores, listas, árvores binárias, etc.

- Existência de rotinas relacionadas: Ao se pesquisar em uma tabela, deve-se saber como esta é criada, como os elementos podem ser inseridos, retirados, etc.

- Independência de representação: Uma estrutura modular verdadeiramente flexível habilita seus clientes a uma operação sem o conhecimento de modo pelo qual o módulo foi implementado. Ou seja, o cliente não deve ter que se importar com qual busca é a melhor para sua entrada, por exemplo, o módulo deve realizar essa verificação internamente. Além disso, os clientes devem ser imunes também a mudanças durante a execução. Exemplificando: se não houver esse mecanismo automático o código deve conter um controle do tipo:
    - se T é do tipo A então “mecanismo A”
    - se T é do tipo B então “mecanismo B”
    - se T é do tipo C então “mecanismo C”
- Semelhanças nos subcasos: Este afeta o design e a construção dos módulos, Ele é fundamental pois determina a possibilidade da construção de módulos bem construídos sem repetições indesejáveis. O aparecimento de repetições excessivas nos módulos compromete suas consistências internas e torna-os difíceis de fazer manutenção. Deve existir um conjunto de soluções já implementadas para serem escolhidas num subcaso.

2. Herança

Em c++, o termo herança se aplica apenas às classes. É com a herança que se pode construir e estender classes desenvolvidas por outras pessoas. Em c++ deve-se fazer classes que resolvam um determinado problema. Quando uma classe é derivada, cria-se uma nova classe que pode herdar algumas ou todas as características da classe pai. Uma das suas características é uma classe poder incorporar comportamentos de todas suas classes bases.

2.1 Classes derivadas

Um exemplo de herança é encontrado a seguir:

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

Nesse exemplo, a classe Caixa é a classe base para a classe CaixaColorida (classe derivada). A classe CaixaColorida herda duas funções e duas variáveis da classe base. Logo, o seguinte código é possível:

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

As funções herdadas não precisam de nenhum tipo de especificação para serem usadas, como visto acima.

2.2 O que não é herdado

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

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

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

2.3 Membros de classes protected

Aqui, tem-se a definição de protected, antes não discutida: ele funciona como private, não é acessível fora da classe; a diferença está na herança. Enquanto um atributo private de uma classe base não é visível na classe derivada, um protected é, e continua sendo protected na classe derivada.

2.4 Construtores e destrutores

Se tratando de uma classe derivada, o construtor da classe base também deve ser chamado. A ordem de chamada dos construtores é fixa. Primeiro a classe base é construída, para depois a derivada ser construída. Se a classe base também deriva de alguma outra, o processo se repete até a classe pai ser alcançada. Os destrutores são chamados na ordem inversa: primeiro, os atributos mais especializados são destruídos, depois os mais gerais.

2.5 Herança pública x herança privada

Os especificadores de acesso private e public podem ser usados na declaração de uma herança; por padrão, as heranças são private. Estes especificadores afetam o nível de acesso que os atributos terão na classe derivada:

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

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