9. HERANÇA MÚLTIPLA

A herança múltipla entre classes ocorre sempre que uma subclasse possui duas ou mais superclasses imediatas, ou seja, é "filha" de mais de uma classe. Através da herança múltipla é possível combinar as características de várias superclasses existentes como um ponto de partida para a definição de uma nova classe. A sintaxe de definição de uma classe com múltiplas superclasses é a seguinte: "*class <nome_classe_derivada> : <especif_acesso> <nome_classe_base1>, <especif_acesso> <nome_classe_base2>, ..., <especif_acesso> <nome_classe_baseN> { //membros da classe };*", onde <*especif_acesso*> pode ser *private*, *public* ou *protected*. Caso não seja colocado nenhum especificador de acesso, assume-se *private* como *default*.

Para exemplificar, considere a definição de uma classe "clock", cujos objetos armazenam o tempo, e uma classe "calendar", cujos objetos armazenam a data. Pode-se então usar a herança múltipla para derivar uma nova classe "clock_calendar", cujos objetos armazenam ambos, o tempo e a data. A figura 10 mostra a hierarquia destas três classes.

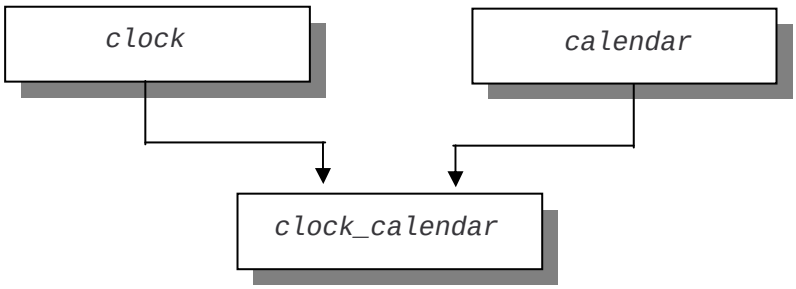

Figura 1 - Exemplo de herança múltipla

A classe "*clock*" é definida da seguinte maneira:

```
class clock {
        protected:
                int hr;
                int min;
                int sec;
                int is_pm;
        public:
                clock(int h, int m, int s, int pm);
                void set_clock (int h, int m, int s, int pm);
                void read_clock (int &h, int &m, int &s, int &pm);
                void advance();
};
```

Nesta classe, as variáveis "hr", "min" e "sec" guardam, respectivamente, as horas, os minutos e os segundos. A variável "is_pm" é 0 para A.M. e 1 para P.M. O construtor "clock()" cria um novo relógio e determina a hora de acordo com os parâmetros recebidos; "set_clock()" seta a hora corrente para a hora passada como parâmetro; "read_clock()" retorna a hora corrente através de seus parâmetros de referência; e "advance()" avança o relógio em um segundo.

A classe "calendar" é similar à "clock":

```
class calendar {
        protected:
                int mo;
                int day;
                int yr;
        public:
                calendar(int m, int d, int y);
                void set_calendar (int m, int d, int y);
                void read_calendar (int &m, int &d, int &y);
                void advance();
};
```

As variáveis "mo", "day" e "yr" guardam a data, isto é, mês, dia e ano, respectivamente. O construtor "calendar()" cria um calendário e determina a data de acordo com os parâmetros recebidos; "set_calendar()" seta um calendário existente para a data passada como parâmetro; "read_calendar()" retorna a data corrente através de seus parâmetros de referência; e "advance()" avança a data em um dia.

Agora pode-se definir a classe "clock_calendar":

```
class clock_calendar : public clock, public calendar {
        public:
                clock_calendar(int mt, int d, int y, int h, int mn, int s, int pm);
                void advance();
};
```

É possível observar que, após o nome da classe e os dois pontos, aparece uma lista de todas as superclasses a partir das quais a classe é derivada. Cada superclasse pode ser especificada individualmente como *public*, *private* ou *protected*. Como é desejado que "clock_calendar" ofereça as funções herdadas de "clock" e "calendar", ambas são declaradas como públicas.

Torna-se importante salientar que, analogamente ao que ocorria na herança simples, a subclasse em uma herança múltipla pode acessar os construtores de suas superclasses. Para efetivar as chamadas, deve-se generalizar a sintaxe vista no estudo de herança simples. A definição do construtor de uma subclasse de mais de uma superclasse pode fazer chamadas a construtores da base. Para tal, o construtor da derivada deve ter como parâmetros os que ele mesmo usa e os que deseja passar aos construtores das bases.

No caso do exemplo que está sendo apresentado, para inicializar o objeto "clock_calendar", o construtor "clock_calendar()" precisa invocar os construtores de "clock" e "calendar". Como na herança simples, isto é feito colocando-se chamadas à "clock()" e "calendar()" na lista de inicialização de "clock_calendar()". Assim sendo, o construtor é definido da seguinte forma:

```
clock_calendar :: clock_calendar ( int mt, int d, int y, int h, int mn, int s, int pm ) : calendar(mt, d, y), clock(h, mn, s, pm)
{ }
```

Infelizmente, o tamanho da lista de argumentos dificulta a leitura da definição. Entretanto, o nome da função qualificada é seguida pela lista de argumentos, e então pela lista de inicialização que chama "clock" e "calendar". Todo trabalho do construtor resume-se em chamar a lista de inicialização: o bloco que define o corpo da função está vazio.

Existe um problema relacionado a ambigüidade em herança múltipla, que ocorre quando as superclasses possuem membros homônimos e a subclasse não redefine esses membros entre suas declarações. Quando um objeto da subclasse tentar referenciar diretamente o membro homônimo das superclasses, o compilador não saberá a qual classe está se referindo, causando a ambigüidade.

Voltando ao exemplo, observa-se que as funções membro "set_clock()", "read_clock()", "set_calendar()" e "read_calendar()" são todas herdadas pela classe "clock_calendar" e trabalham exatamente da mesma maneira tanto para os objetos de "clock_calendar" como para "clock" ou "calendar". Entretanto, todas as três classes possuem uma função "advance()"; a versão de "advance()" declarada em "clock_calendar" sobrescreve aquela herdada de "clock()" e "calendar()". Entretanto, as funções herdadas "clock::advance()" e "calendar::advance()" podem ser usadas na definição de "clock_calendar::advance()":

```
void clock_calendar :: advance() {
        int was_pm = is_pm;
        clock::advance();
        if ( was_pm && !is_pm )
                calendar::advance();
}
```

Esta função chama "clock::advance()" para avançar o relógio; se o avanço do relógio muda de P.M. para A.M., então "calendar::advance()" é chamada para avançar o calendário.

Se as versões de "advance()" não fossem sobrescritas pela declaração "advance()" em "clock_calendar", se "cc", por exemplo, é um objeto de "clock_calendar", o comando:

```
cc.advance();
```

é ambíguo (e portanto inválido) porque "advance()" pode se referir a "clock::advance()" ou "calendar::advance()", ambos podendo ser herdados como membros públicos de "clock_calendar". Assim, deve-se escrever também:

```
cc.clock::advance();
```

para avançar a parte do relógio de "cc", ou

```
cc.calendar::advance();
```

para avançar a parte do calendário.

Normalmente, é indesejável exigir que usuários empreguem tais nomes complexos. A melhor solução é o que realmente foi feito: fornecer à "clock_calendar" uma função "advance()" que sobrescreve as funções "advance()" herdadas e cuja definição chama-as conforme o necessário para avançar o objeto "clock_calendar" [3, 9].

Existe um segundo tipo de ambigüidade que pode surgir em herança múltipla que está relacionado à forma como a hierarquia foi definida. Dependendo de como as classes se relacionam, pode ocorrer uma duplicidade de atributos nas superclasses. Por exemplo, na figura 11, define-se uma hierarquia com uma superclasse "ClasA", duas classes derivadas, "ClasB1" e "ClasB2", gerando a classe de nome "ClasC". Um possível código C++ para as classes também é apresentado.

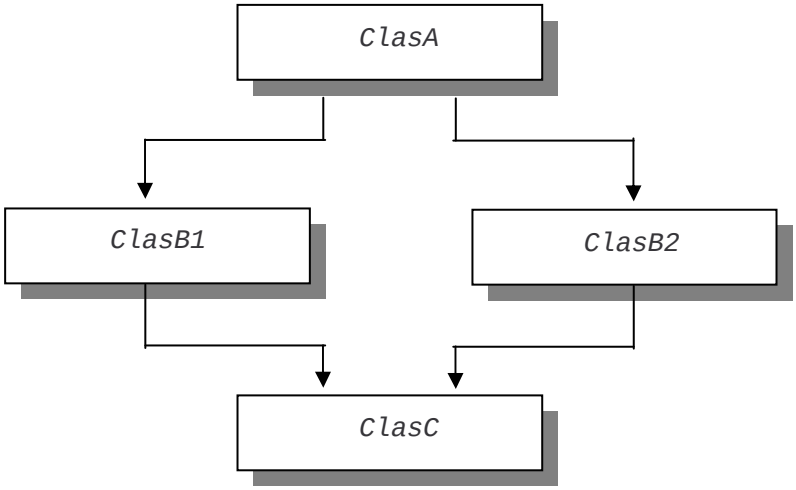

Figura 2 - Hierarquia com possível duplicidade de membros na superclasse

```
#include <string.h>
#include <iostream>
using namespace std;
class ClasA {
        protected :
                char nome[31];
        public :
                void mostrar () {cout << "\nNome na Classe\t: " << nome;}
};
class ClasB1 : public ClasA {
        public :
                ClasB1 (char * n = "Classe B1") {strcpy(nome,n);}
                void mostrar() {ClasA::mostrar();}
};
class ClasB2 : public ClasA {
        public :
                ClasB2(char * n = "Classe B2") {strcpy(nome,n);}
                void mostrar() {ClasA::mostrar();}
};
class ClasC : public ClasB1, public ClasB2 {
        public :
                void saida () {
                        ClasB1::mostrar();
                        ClasB2::mostrar();
                }
};
```

```
void main () {
        clrscr();
        ClasC objClasC;
        objClasC.saida();
}
```

O problema que deve ser considerado neste tipo de hierarquia é que a criação de um objeto da classe "ClasC" pode causar uma duplicidade de atributos na superclasse "ClasA". Um dos objetos de "ClasA" é criado pela classe "ClasB1", e outro pela classe "ClasB2". Para mostrar esta duplicidade, o programa apresentado simplesmente imprime o valor de uma atributo da classe "ClasA" (*string* "nome"), uma vez através da classe "ClasB1" e outra através da classe "ClasB2", cujas inicializações seguem valores *default* diferentes. Os resultados do programa identificarão a duplicidade, uma vez que imprimem o seguinte na tela:

```
Nome na Classe: Classe B1
Nome na Classe: Classe B2
```

Resumindo a duplicidade significa a existência de dois espaços em memória para um mesmo atributo ou a existência de dois caminhos para a chamada de uma função. Inicialmente pode-se achar a duplicidade de atributos inadequada, mas ela pode ser útil quando uma subclasse usa um campo com significado dependente do caminho que usa para alcançá-lo. Por exemplo, a data pode ser um atributo que significa "dia de pagamento" a um fornecedor e "dia de recebimento" a um cliente. Quando for ambos, pode-se usar o campo "data" com duplo sentido.

Quando a duplicidade for indesejável, deve-se lançar mão de outro recurso, a superclasse virtual. A declaração virtual para uma classe soluciona a ambigüidade da duplicidade de atributos simplesmente eliminando a criação de dois endereços para os membros na superclasse da hierarquia. Sabe-se que membros estáticos em classes são compartilhados por todos os objetos. Em uma analogia, a superclasse virtual provoca o mesmo efeito para as instâncias superiores na hierarquia da herança múltipla. Assim, no programa apresentado, acessar o membro "nome" da classe "ClasA" através do caminho "ClasC-ClasB1-ClasA" é o mesmo que fazê-lo percorrendo "ClasC-ClasB2-ClasA".

A declaração de que a superclasse é virtual se dá na subclasse, colocando-se a palavra reservada virtual entre o especificador de acesso e o nome da superclasse. Para ilustrar, no exemplo anterior pode-se declarar a classe "ClasA" como virtual modificando as definições de "ClasB1" e "ClasB2" para:

```
class ClasB1 : public virtual ClasA {
        public :
                ClasB1 (char * n = "Classe B1") {strcpy(nome,n);}
                void mostrar() {ClasA::mostrar();}
};
class ClasB2 : public virtual ClasA {
        public :
                ClasB2(char * n = "Classe B2") {strcpy(nome,n);}
                void mostrar() {ClasA::mostrar();}
};
```

Com essas novas classes, o programa apresentaria a seguinte saída:

```
Nome na Classe: Classe B2
Nome na Classe: Classe B2
```

A última classe na derivação da classe "ClasC", ou seja, "ClasB2", é quem coloca o valor definitivo para o atributo nome.

Deve-se notar que as declarações virtual devem estar presentes em ambas as classes derivadas da raiz "ClasA". Também não adianta colocar a declaração virtual na definição da classe "ClasC", mesmo que se faça isto para ambas as classes ("ClasB1" e "ClasB2") [9].