

Capítulo 2b

Recursos de C++ relacionados às classes

Classes aninhadas

Declarações de classes podem ser aninhadas. Uma classe aninhada não reside no espaço de nomes global, como era em C. Para referenciar uma classe aninhada fora do seu escopo, é preciso usar o operador de escopo.

Este aninhamento é usado para encapsular classes auxiliares que só fazem sentido dentro de um escopo reduzido.

```
struct A {
    struct B {
        int b;
    };
    int a;
};

void main()
{
    A a;
    A::B b;
}
```

O código acima simplesmente declara a estrutura B dentro de A; não existe nenhum campo de A que seja do tipo B.

Declaração incompleta

Em C, a declaração de estruturas mutuamente dependentes é feita naturalmente:

```
struct A { struct B *next; };
struct B { struct A *next; };
```

Em C++ a construção acima pode ser usada. No entanto, como a declaração de uma estrutura em C++ já define o nome como um tipo sem necessidade de typedefs, é normal em C++ o uso de tipos sem o uso da palavra reservada struct:

```
struct A { B *next; }; // Erro! B indefinido
struct B { A *next; };
```

No caso de tipos mutuamente dependentes, isto só é possível usando uma declaração incompleta:

```
struct B; // declaração incompleta de B
struct A { B *next; };
struct B { A *next; };
```

Assim como em C, antes da declaração completa da estrutura só é possível usar o nome para declarar ponteiros e referências deste tipo. Como as informações estão incompletas, não é possível declarar variáveis deste tipo nem utilizar campos da estrutura.

Métodos const

Métodos const são métodos que não alteram o estado interno de um objeto. Assim como era possível declarar variáveis e parâmetros const em C, é possível declarar um método const em C++. A especificação const faz parte da declaração do método.

A motivação para esta nova declaração pode ser esclarecida com o exemplo abaixo:

```
struct A {
    int value;
    int get()      { return value; }
    void put (int v) { value = v; }
};
```

```

void f(const A a)
{
    // parâmetro a não pode ser modificado
    // quais métodos de a podem ser chamados???
    int b = a.get(); // Erro! método não é const
}

```

A função *f* não pode alterar o seu parâmetro por causa da declaração *const*. Isto significa que os campos de *a* podem ser lidos mas não podem ser modificados. E quanto aos métodos? O que define quais métodos podem ser chamados é a sua implementação. Esta verificação não pode ser feita pelo compilador, já que na maioria das vezes o código dos métodos não está disponível. É responsabilidade do programador declarar quais métodos não modificam o estado interno do objeto. Estes métodos podem ser aplicados a objetos declarados *const*.

No exemplo acima, o método *get* pode ser declarado *const*, o que possibilita a sua chamada na função *f*. Se o método *put* for declarado *const*, o compilador acusa um erro de compilação pois o campo *value* é modificado em sua implementação:

```

struct A {
    int value;
    int get() const { return value; }
    void put (int v) { value = v; }
};

void f(const A a)
{
    // parâmetro a não pode ser modificado
    int b = a.get(); // ok, método const
}

```

this

Em todo método não *static* (ver seção sobre métodos *static*), a palavra reservada *this* é um ponteiro para o objeto sobre o qual o método está executando.

Todos os métodos de uma classe são sempre chamados associados a um objeto. Durante a execução de um método, os campos do objeto são manipulados normalmente, sem necessidade de referência ao objeto. E se um método precisar acessar não os campos de um objeto, mas o próprio objeto? O exemplo abaixo ilustra este caso:

```

struct A {
    int i;
    A& inc();
};

// Este método incrementa o valor interno
// e retorna o próprio objeto
A& A::inc()
{
    // estamos executando este código para obj1 ou obj2?
    // como retornar o próprio objeto?
    i++;
    return *this; // this aponta para o próprio objeto
}

void main()
{
    A obj1, obj2;
    obj1.i = 0;
    obj2.i = 100;
    for (j=0; j<10; j++)

```

```
        printf("%d\n", (obj1.inc()).i);
    }
```

O tipo de `this` dentro de um método de uma classe `X` é

```
X* const
```

a não ser que o método seja declarado `const`. Nesse caso, o tipo de `this` é:

```
const X* const
```

Campos de estrutura static

Em C++, membros de uma classe podem ser `static`. Quando uma variável é declarada `static` dentro de uma classe, todas as instâncias de objetos desta classe compartilham a mesma variável. Uma variável `static` é uma variável global, só que com escopo limitado à classe.

A declaração de um campo `static` aparece na declaração da classe, junto com todos os outros campos. Como a declaração de uma classe é normalmente incluída em vários módulos de uma mesma aplicação via *header files* (arquivos `.h`), a declaração dentro da classe é equivalente a uma declaração de uma variável global `extern`. Ou seja, a declaração apenas diz que a variável existe; algum módulo precisa defini-la. O exemplo abaixo ilustra a utilização de campos `static`:

```
struct A {
    int a;
    static int b; // declara a variável,
                  // equivalente ao uso de extern para
                  // variáveis globais
};

int A::b = 0;     // define a variável criando o seu espaço
                  // esta é a hora de inicializar

void main(void)
{
    A a1, a2;
    a1.a = 0; // modifica o campo a de a1
    a1.b = 1; // modifica o campo b compartilhado por a1 e a2
    a2.a = 2; // modifica o campo a de a1
    a2.b = 3; // modifica o campo b compartilhado por a1 e a2
    printf("%d %d %d %d", a1.a, a1.b, a2.a, a2.b);
    // imprime 0 3 2 3
}
```

Se a definição

```
int A::b = 0;
```

for omitida, o arquivo é compilado mas o linker acusa um erro de símbolo não definido.

Como uma variável estática é única para todos os objetos da classe, não é necessário um objeto para referenciar este campo. Isto pode ser feito com o operador de escopo:

```
A::b = 4;
```

Métodos static

Assim como campos `static` são como variáveis globais com escopo reduzido à classe, métodos `static` são como funções globais com escopo reduzido à classe. Isto significa que métodos `static` não tem o parâmetro implícito que indica o objeto sobre o qual o método está sendo executado (`this`), e portanto apenas os campos `static` podem ser acessados:

```
struct A {
    int a;
    static int b;
    static void f();
};
```

```

int A::b = 10;

void A::f()
{
    a = 10; // errado, a só faz sentido com um objeto
    b = 10; // ok, b declarado static
}

```

Ponteiros para métodos

É possível obter o ponteiro para um método. Isto pode ser feito aplicando o operador & a um nome de método totalmente qualificado, ou seja:

& classe::método

Uma variável do tipo ponteiro para um membro da classe X é obtida com o declarador X::*:

```

struct A {
    void f(int);
};

void main()
{
    void (A::*p) (int); // declara a variável p
    p = &A::f;          // obtém o ponteiro para f

    A a1;
    A* a2 = new A;

    (a1.*p)(10);          // aplica p a a1
    (a2->*p)(20);          // aplica p a a2
}

```

A aplicação dos métodos é feita utilizando-se os novos operadores .* e ->.*.

