

Programação C++ para Jogos II



Índice - Apresentação II

Membros estáticos	3
Herança	7
Herança Múltipla.....	10
Polimorfismo	12
Herança com ponteiros	14
Problema do Diamante	17
Classes Abstratas	19
Interfaces	21
Redefinição de Operadores	23

Membros Estáticos

Até agora, as propriedades e métodos dos exemplos sempre foram distintos para cada instância. Por exemplo, cada `Pessoa` instanciada tem a sua própria idade, o seu próprio peso e o seu próprio nome. Estes podem se repetir eventualmente, mas não são comuns a todas as pessoas.

Mas existem características que são comuns a todos os objetos de uma classe, sem exceções. Por exemplo, todas as instâncias da nossa classe `Dado` possuem seis lados.

Uma `pessoa` pode se chamar Maria e outra se chamar João. Por isso, `nome` é uma de propriedade não estática.

Mas para um `dado`, o `número de lados` é comum a todos os dados de um mesmo gênero, por isso este atributo, quando presente, deve ser criado como atributo **static**.

A classe `Dado` será reescrita no próximo slide, incluindo o novo atributo **numeroLados**.

```

#include <iostream>
#include <time.h>
using namespace std;

class Dado {
private:
    static int numeroLados;
    int valor;
public:
    inline int getValor(void) {
        return valor;
    }
    inline static void setNumeroLados(int lados) {
        numeroLados = lados;
    }
    int sorteia(void) {
        return valor = (rand() % numeroLados) + 1;
    }
    Dado() {
        srand (time(NULL));
        valor = 0;
    }
    ~Dado(void) {}
};

int Dado::numeroLados = 6;

int main(void) {
    Dado::setNumeroLados(20);
    Dado dado1,dado2;
    cout << dado1.sorteia() << endl;
    cout << dado2.sorteia() << endl;
    cout << dado1.getValor() + dado2.getValor() << endl;
    return 0;
}

```

- O valor de um atributo estático é modificado na classe. No exemplo anterior, todos as instâncias foram transformadas em Dados de 20 lados.

- Como o atributo numeroLados pode receber um eventual tratamento (não aceitar mais de 100 lados, por exemplo), ele foi encapsulado. Veja que ele é acessado apenas por um Set.

- Todas as propriedades estáticas precisam ser setadas do lado de fora da classe, como foi feito no programa anterior. Esta chamada inicial externa deve ser no formato:

```
Tipo NomeClasse::Propriedade = Valor;
```

- Atributos e Métodos são chamados de suas instâncias com .(ponto) ou ->(seta). Mas atributos estáticos são chamados diretamente da classe com ::. São independentes da existência de instâncias, como visto na primeira linha da função Main.

- ATENÇÃO: Atributos estáticos podem também ser chamadas pelas instâncias, mas seus valores serão modificados em todas as outras instâncias. Ou seja, serão modificadas na classe. Veja outro exemplo no próximo slide.

```

#include <iostream>
using namespace std;

class X
{
public:
    static int a;
    int b;

    X(int b) : b(b) {}

    static int f(int c) {
        return c * c;
    }

    void print(void) {
        cout << b << endl;
    }
};

int X::a = 0;

int main(void)
{
    X x1(3);
    X x2(8);
    x1.print();
    x2.print();
    x1.a = 18;
    x2.a = 81;
    // x1.a vale 81.
    cout << x1.a << endl;
    cout << X::a << endl;
    cout << X::f(3) << endl;
    return 0;
}

```

Herança (Derivação)

Com a Orientação a Objetos podemos simular o mundo real num programa, deixando-o mais organizado e legível.

Suponha que em seu projeto seja necessário criar uma classe **Professor**. Esta classe possui algumas características que só um professor tem: Salário, Número de Alunos... Mas algumas características de **Professor** são comuns a qualquer outra **Pessoa**: nome, idade, CPF, estado civil, endereço... Quando estas características em comuns são detectadas, é comum que se use uma ferramenta de Orientação a Objetos chamada **Herança**.

Para este exemplo, deve ser criada uma classe Pessoa com as características de uma pessoa e uma classe Professor com as características que APENAS professores possuem (aqui não entrariam nome, idade, ...).

Feitas as duas classes, define-se que **Professor** herda de **Pessoa**, o que significa que agora todas as instâncias de Professor possuem também características presentes apenas na classe Pessoa: nome, cpf, idade, etc.

Para que a idéia fique mais clara, veja um exemplo no próximo slide.

```

#include <iostream>
using namespace std;

class Pessoa
{
private:
    string _nome;
    short _idade;
public:
    Pessoa(string nome, short idade) : _nome(nome), _idade(idade) {}

    inline string getNome (void)      { return this->_nome; }
    inline void  setNome  (string nome) { this->_nome = nome; }
    inline short  getIdade (void)      { return this->_idade; }
    inline void  setIdade (short idade) { this->_idade = idade; }
};

class Professor : public Pessoa
{
private:
    int _salario;
    int _nAlunos;
public:
    Professor(string nome, short idade, int salario, int nAlunos) :
        Pessoa(nome, idade), _salario(salario), _nAlunos(nAlunos) {}

    inline int getSalario (void)      { return this->_salario; }
    inline void setSalario (int salario) { this->_salario = salario; }
    inline int getNAlunos (void)      { return this->_nAlunos; }
    inline void setNAlunos (int nAlunos) { this->_nAlunos = nAlunos; }
};

int main(void)
{
    Professor p("Maria", 32, 1000, 69);
    cout << "Nome: " << p.getNome() << endl;
    cout << "Idade: " << p.getIdade() << endl;
    cout << "Salario: " << p.getSalario() << endl;
    cout << "Alunos: " << p.getNAlunos() << endl;
    return 0;
}

```

- Classes que tem seus membros herdados são chamadas de classe **pai**, e classes que herdam membros de outras classes são chamadas de classe **filha**.
- Não existem atributos nome e idade dentro da classe **Professor**, e mesmo assim a instância de professor as possui. Isto aconteceu porque **Professor** **herdou** as características de **Pessoa**.
- Uma herança é definida na primeira linha da classe, com o operador **:(dois pontos)**, a palavra **public** e o **nome da classe** de onde virá a derivação.
- Ao criarmos uma instância de **Professor**, indiretamente também estamos criando uma instância de **Pessoa**. Por isso, o construtor de **Pessoa** deve ser chamado junto ao construtor de **Professor**. Como regra geral: **Uma classe filha deve ter, junto de seu construtor, uma chamada do construtor da classe pai**.
- A partir de agora, se quisermos criar qualquer outra classe que tenha relação com uma pessoa (**Aluno**, **Funcionário**, **Jogador**), podemos usar a classe **Pessoa** como base, e assim economizar linhas de código.

Herança Múltipla

No exemplo anterior, a classe Professor herda de **apenas uma** outra, Pessoa. Isto é chamado de herança simples. C++ e poucas outras linguagens permitem que suas classes herdem de **qualquer quantidade** de outras classes. Esta característica se chama herança múltipla.

A herança múltipla pode tornar seu programa confuso e ambíguo, e por isso deve ser evitado sempre que possível. O uso de herança múltipla pode resultar no problema do diamante, que será estudado mais a frente neste material.

Linguagens como Java não permitem este tipo de herança, mas a simulam com uma ferramenta chamada Interface. Estas não existem em C++, mas podem ser simuladas. Interfaces também serão estudadas em breve.



```

#include <iostream>
using namespace std;

class Liquido {
private:
    double _densidade;
public:
    Liquido(double densidade) : _densidade(densidade) {}
    inline double getDensidade(void) { return this->_densidade; }
    inline void setDensidade(double densidade) { this->_densidade = densidade; }
};

class Ingerivel {
private:
    bool _estragado;
public:
    Ingerivel(bool estragado) : _estragado(estrugado) {}
    inline bool getEstrugado(void) { return this->_estragado; }
    inline void trocaEstrugado(void) { this->_estragado = !this->_estragado; }
};

class Agua : public Liquido, public Ingerivel {
private:
    string _origem;
public:
    Agua(double densidade, bool estrugado, string origem)
    : Liquido(densidade), Ingerivel(estrugado), _origem(origem) {}
    inline string getOrigem(void) { return this->_origem; }
    inline void setOrigem(string origem) { this->_origem = origem; }
};

int main(void) {
    Agua a(1,true,"Vala");
    cout << "Densidade: " << a.getDensidade() << endl;
    cout << "Estrugado: " << (a.getEstrugado() ? "Sim" : "Nao") << endl;
    cout << "Origem: " << a.getOrigem() << endl;
    return 0;
}

```

Polimorfismo

Foi visto nos slides anteriores que com herança, a classe filha ganha os métodos e os atributos da classe pai. Mas em alguns casos, queremos que um método herdado funcione de modo diferente nos filhos. É possível reescrever um método herdado, e esta possibilidade é chamada de **Polimorfismo**. É similar a uma sobrecarga de métodos, mas em classes diferentes. Sobrecargas são tipos específicos de polimorfismo.

OBS: Prometi no slide 26 da primeira parte deste material que em breve falaria de um terceiro tipo de escopo, **protected**.

Nos dois programas anteriores as classes estão herdando apenas os membros públicos de seus pais. Embora Professor possua idade e Água possua densidade, estes atributos não são acessíveis diretamente, pois seus escopos são **private**.

Para que um atributo ou método seja privado para qualquer classe exceto suas filhas, o escopo **protected** deve ser usado.



```

#include <iostream>
using namespace std;

class Pessoa {
protected: // '_idade' será usado em Crianca, pois seu escopo é protected
    string _nome;
    short _idade;
public:
    Pessoa(string nome, short idade) : _nome(nome) { setIdade(idade); }

    inline string getNome (void)          { return this->_nome;    }
    inline void   setNome  (string nome)   { this->_nome = nome;    }
    inline short  getIdade (void)          { return this->_idade;  }
    inline void   setIdade (short idade) {
        _idade = idade < 0 ? -idade : idade;
    }
};

class Crianca : public Pessoa {
public:
    Crianca(string nome,short idade) : Pessoa(nome,0) { setIdade(idade); }

    // Uma Pessoa pode ter qualquer idade positiva,
    // mas uma criança pode ter apenas entre 0 e 9 anos.
    inline void setIdade(short idade) {
        this->_idade = idade % 10;
    }
};

int main(void)
{
    Pessoa p("Maria",-30);
    Crianca c("Mario",13);
    cout << p.getIdade() << endl;
    cout << c.getIdade() << endl;
    return 0;
}

```



Herança com Ponteiros

O comportamento da herança é diferente quando usamos uma variável comum e quando usamos um ponteiro. Por enquanto, nossas instâncias nos exemplos de herança e de polimorfismo não foram ponteiros.

Voltando ao exemplo Pessoa/Professor: Nossa classe Professor herda de Pessoa. Se quisermos uma instância de professor referenciada por um ponteiro, poderíamos fazer assim:

```
Professor p = new Professor(...);
```

Porém, é uma boa prática de programação fazer o ponteiro sempre do tipo mais acima. Neste caso, o tipo mais acima é Pessoa:

```
Pessoa p = new Professor(...);
```

Embora seja estranha, esta forma de escrever não é incorreta. E é preferível programar assim, pois numa parte do programa a Pessoa em questão pode ser um Professor, e em outro momento, pode ser um Dançarino.

Em um contexto de jogo, podemos ter um ponteiro para 'Inimigo'. Num momento do jogo este inimigo pode ser comum, e em outro momento, o mesmo ponteiro pode referenciar um chefe.



A herança com ponteiros modifica o funcionamento do polimorfismo. Os métodos reescritos nos filhos são ignorados. Não adianta reescrevê-los, o método do pai sempre é chamado no lugar, pois é levado em consideração não o tipo do objeto, mas o tipo do ponteiro.

Para que isto não aconteça, os métodos que serão reescritos devem ser declarados como **virtuais**. Se um método é virtual, o programa levará em conta o tipo da instância, não o tipo do ponteiro.

Métodos virtuais são um pouco mais lentos do que métodos comuns, por isso use-os apenas quando for necessário reescrevê-los.

Em algumas linguagens, como Java, todos os métodos sempre são virtuais...

ATENÇÃO: Se sua classe possuir pelo menos um método virtual, o **destrutor** deve também ser declarado como **virtual**.



```

#include <iostream>
using namespace std;

// Analise a saída deste programa,
// tendo como base o que foi dito no slide anterior.
class A {
public:
    virtual void f(void) {
        cout << "metodo virtual f() de A" << endl;
    }
    void g(void) {
        cout << "metodo nao-virtual g() de A" << endl;
    }
    virtual ~A() {}
};

class B : public A {
public:
    void f(void) { // O método f() é virtual em A, e foi reescrito em B
        cout << "metodo virtual f() de B" << endl;
    }
    void g(void) {
        cout << "metodo nao-virtual g() de B" << endl;
    }
};

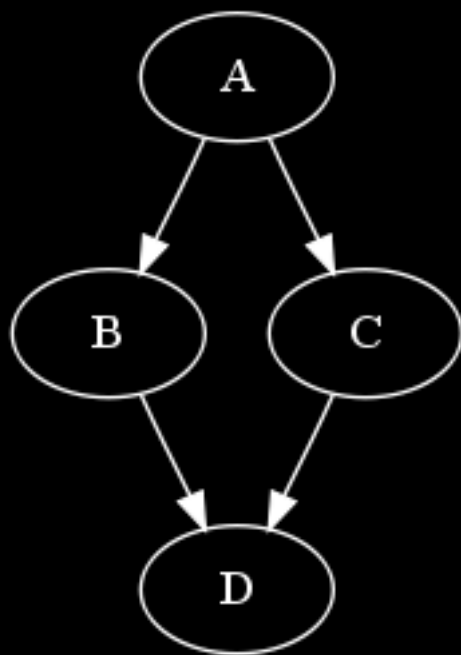
int main(void)
{
    A *a = new A();
    B *b = new B();
    A *c = new B();
    a->f(); a->g(); cout << endl; // Usa apenas os métodos de A
    b->f(); b->g(); cout << endl; // Usa apenas os métodos de B
    c->f(); c->g(); // Usa o virtual de B e o não virtual de A
    return 0;
}

```



Problema do Diamante

Um sistema que possua heranças múltiplas, mesmo se bem modelado, pode eventualmente cair no chamado Problema do Diamante. Apesar do nome estranho, a idéia é bastante simples.



Suponha que cada letra do desenho ao lado represente uma classe.

A classe **A** possui um atributo **x**. As classes **B** e **C** são filhas de **A**, e por isso também possuem o atributo **x**.

Agora, por **herança múltipla**, a classe **D** herda de **B** e de **C** ao mesmo tempo. Isto faz com que **D** possua o atributo **x** duas vezes. Este problema pode ser resolvido com a chamada **Herança Virtual**. Este tipo de herança ignora um dos atributos, em caso de repetição.

```
#include <iostream>
using namespace std;

class A
{
    public:
        int x;
};

// Ambos B e C, embora vazios,
// possuem o atributo x, de A.
class B : public virtual A { };
class C : public virtual A { };

// D possuiria x duas vezes,
// se não fosse a herança virtual.
class D : public B, public C { };

int main(void)
{
    D d;
    d.x = 3;
    cout << d.x << endl;

    return 0;
}
```



Classes Abstratas

Classes abstratas são tipos especiais de classes que obrigam seus filhos a implementar certos métodos.

Em alguns casos específicos sabemos que uma classe possui um método, mas por ela ser muito **genérica**, este método não é capaz de ser implementado. Estes **métodos** que não conseguimos implementar são chamados de **abstratos**.

Exemplo clássico: Sua modelagem possui uma classe FiguraGeometrica. Toda figura geométrica possui área, mas como implementaríamos um método area(), pertencente a esta classe? Não é possível, pois não sabemos a forma exata desta figura. Ela é apenas uma idéia genérica. Neste exemplo, o método area() é abstrato.

Por herança, poderíamos criar classes Círculo e Retângulo. Estes sim podem ter o método area() facilmente implementados.

Observações:

- Uma classe é Abstrata se tiver **ao menos um** método abstrato.
- Classes Abstratas **não podem** ser instanciadas. (Por quê?!)
- Se uma subclasse de uma classe abstrata não implementar os métodos abstratos, ela também será considerada abstrata.
- Classes Abstratas **podem ter** atributos e métodos "comuns".
- Métodos Abstratos são **igualados a zero**, e não possuem corpo.
- Métodos Abstratos são **virtuais**. (Por quê?!)



```

#include <iostream>
using namespace std;

class FiguraGeometrica {
protected:
    bool _convexo; // Forcei a barra, só pra ter um atributo.
public:
    virtual double area(void) = 0;
    virtual double perimetro(void) = 0;

    FiguraGeometrica(bool convexo) : _convexo(convexo) {}
};

class Circulo : public FiguraGeometrica {
private:
    double raio;
public:
    Circulo(double raio) : FiguraGeometrica(true) , raio(raio) {}
    inline double area(void) { return 3.1416 * raio * raio; }
    inline double perimetro(void) { return 2 * 3.1416 * raio; }
};

class Quadrado : public FiguraGeometrica {
private:
    double lado;
public:
    Quadrado(double lado) : FiguraGeometrica(true) , lado(lado) {}
    inline double area(void) { return lado * lado; }
    inline double perimetro(void) { return 4 * lado; }
};

int main(void) {
    // A seguinte linha, comentada, não compilaria.
    // FiguraGeometrica *fg = new FiguraGeometrica;
    FiguraGeometrica *c = new Circulo(1.0);
    FiguraGeometrica *q = new Quadrado(5.0);
    cout << c->area() << endl;
    cout << c->perimetro() << endl;
    cout << q->area() << endl;
    cout << q->perimetro() << endl;

    return 0;
}

```



Interfaces

Como vimos no slide 17, heranças múltiplas podem gerar o problema do diamante. E este ocorre por causa de membros duplicados.

Em outras linguagens, como `Java`, este problema nunca vai acontecer. Java baniu a herança múltipla, mas ainda é capaz de simulá-la, com uma ferramenta chamada `Interface`.

Classes Java podem herdar de `apenas UMA` outra classe, mas são livres para `implementar` qualquer quantidade de Interfaces.

Interfaces são classes nas quais todos os métodos são abstratos. E estas também não possuem atributos.

Atenção para a nomenclatura: classes são `herdadas`, interfaces são `implementadas`.

O uso de Interfaces exige um cuidado maior com a sua modelagem, mas acaba de vez com o `Problema do Diamante`.

Interfaces não existem oficialmente em `C++`, mas podem ser simuladas, como é visto no próximo exemplo.



```

#include <iostream>
using namespace std;

class A {
    public:
        int x;
};

// B possui o atributo x, de A.
class B : public A { };

// C não pode herdar de A. Se herdasse, possuiria o atributo x,
// e assim deixaria de ser uma Interface.
class C {
    public:
        virtual void f(void) = 0;
};

class D : public B, public C
{
    public:
        void f(void)
        {
            cout << "Alguma coisa" << endl;
        }
};

int main(void)
{
    D d;
    d.x = 3;
    cout << d.x << endl;
    d.f();

    return 0;
}

```



Redefinição de Operadores

Para qualquer linguagem de programação, estamos acostumados apenas a decorar o que faz cada operador. Por exemplo, que o `+` soma, que o `*` multiplica e que o `[]` acessa posições.

Mas em C++ temos também a liberdade de decidir o que cada operador pode fazer. Qualquer operador (dentro os redefiníveis) pode fazer qualquer coisa.

Operadores em C++ podem ser tratados como funções. O operador `=`, se redefinido numa classe, pode ser acessado de um destes dois modos:

```
A = B;  
A.operator=(B);
```

E não só o `=`, mas qualquer operador redefinível pode ser usado como uma função `'operator'`. Para editar o que um operador faz, precisamos reescrever a função `'operator'` específica com o novo comportamento desejado.

Operadores que aceitam redefinição: `, ! != % %= & && &= () * *= + ++ += - -- -= -> ->* / /= < << <<= <= = == > >= >> >>= [] ^ ^= | |= || ~ delete new.`



```

#include <iostream>
using namespace std;

class Inteiro {
private:
    int _val;
public:
    Inteiro(int val = 0) {
        this->val() = val;
    }
    inline int &val() { return _val; }

    Inteiro operator+(Inteiro i) {
        i.val() += val();
        return i;
    }
    // Pode ser feito sobrecarga
    Inteiro operator+(int i) {
        i += val();
        return i;
    }
    // Atenção com o formato deste método
    Inteiro& operator+=(Inteiro i) {
        val() += i.val();
        return *this;
    }

    Inteiro& operator+=(int i) {
        val() += i;
        return *this;
    }
};

```

Este exemplo redefine + e +=. O formato dos métodos dos operadores -, *, /, %, -=, *=, /= e %= segue o mesmo formato.

```

// Embora os operadores sejam
// usados por variáveis do tipo
// Inteiro, internamente as contas
// são feitas no do atributo _val
int main(void)
{
    Inteiro i(8);
    Inteiro j(2);
    Inteiro k;

    k = i + 3; // k = 11
    k = k + j; // k = 13
    k += 9;    // k = 22
    k += i;    // k = 30
    cout << k.val() << endl;
    return 0;
}

```



O `operador -` pode ser redefinido de duas formas diferentes.

1) `operator-(parâmetro)` faz uma operação qualquer, tendo como base um outro valor. Assim como foi visto no slide anterior com o `operator+`. Exemplo de uso: `i = i - 2;`

2) `operator-()`, sem parâmetro, supõe que estamos interessados no sinal de menos usado na trocar de sinal. Exemplo de uso: `-i;`

A redefinição deste operador, para a classe `Inteiro`, ficaria assim:

```
Inteiro operator-()
{
    return -val();
}
```



O `operador =` tem sua construção similar aos operadores `+=` e `-=`. Estes operadores modificam os `próprios valores` dos objetos, e por isso, devem retornar uma referência a si mesmos após a automodificação.

O operador `=` também serve para modificar o próprio objeto. O objeto em questão passa a ser igual a um outro valor. Por isso deve também retornar uma referência para si, após as modificações necessárias.

Esta é uma possível redefinição para a classe `Inteiro`:

```
Inteiro& operador=(const Inteiro& i)
{
    _val = i._val;
    return *this;
}
```



Antes de mostrar a redefinição dos operadores ++ e --, é importante lembrar a diferença entre o pré e o pós incremento/decremento.

Para variáveis int, caso seja feito um pós incremento:

```
i = 1;  
j = 2;  
i = j++;
```

O i receberia o valor de j antes do incremento. Para este trecho, ***i vale 2 e j vale 3.***

E para o pré incremento:

```
i = 1;  
j = 2;  
i = ++j;
```

Primeiro é feito o incremento, e só depois a atribuição. Neste caso, ***i vale 3 e j vale 3.***



```
// Possíveis redefinições do ++
// para a classe Inteiro.
// O mesmo vale para o --.

//Para ++i
Inteiro& operator++()
{
    ++val();
    return *this;
}

//Para i++
Inteiro operator++(int)
{
    int v = val();
    val()++;
    Inteiro i(v);
    return i;
}
```



```
// Possível redefinição do operador ^,  
// funcionando como expoente.  
  
Inteiro operator^(Inteiro exp)  
{  
    int i = (int) pow((double)val(), (double)exp.val());  
    return i;  
}
```

O operador `^` normalmente é usado como XOR. Para A e B valores inteiros, **`A ^ B` é `A XOR B`**.

Esta redefinição modifica completamente o significado do operador `^` para a classe Inteiro. Agora `i = i^2` retornaria 64, para `i` inicialmente valendo 8.

Atenção: A função acima deveria retornar um Inteiro, mas está retornando um `int`. Isto é possível, pois existe um **construtor** na classe Inteiro que recebe um `int` como parâmetro.



Os operadores `<`, `<=`, `>`, `>=`, `==`, `!=`, `&&` e `||` naturalmente retornam `bool` e são usados dentro de condições. Caso o mesmo efeito seja desejado na classe `Inteiro`, esta seria uma possível redefinição:
(Mostrarei apenas para um dos operadores. Os outros são análogos.)

```
bool operator<(Inteiro i)
{
    if(val() < i.val())
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

A redefinição de operadores não precisa acontecer apenas dentro de classes. No próximo exemplo, o operador << está redefinido para listar os elementos de um vetor de int, caso este vetor seja usado em conjunto com cout.

```
#include <iostream>
using namespace std;

// Para facilitar a nossa(minha) vida, vou supor que
// o vetor sempre tem tamanho 3.
// Geralmente, dentro de classes, operadores recebem um ou nenhum parâmetro.
// E geralmente, fora de classes, operadores recebem um ou dois parâmetros.
ostream& operator<< ( ostream& o, int* vetor )
{
    o << vetor[0] << " " << vetor[1] << " " << vetor[2];
    return o;
}

int main(void)
{
    int vetor[3] = { 1 , 4 , 3 };
    cout << vetor << endl;
    return 0;
}
```



operator() , objeto-função

Um dos operadores mais importantes é o (), Parêntesis. A redefinição deste operador faz com que um objeto possa ser usado como se fosse uma função. No próximo exemplo, o operador parêntesis está redefinido para funcionar como um Get da classe Inteiro. Este operador pode receber qualquer quantidade de parâmetros.

```
int operator() (void)
{
    return _val;
}
```

```
//... E na main...
Inteiro i(3);
cout << i() << endl;
```



```

//...
//... Continuação da classe Inteiro.

// Esta redefinição retorna o dobro do valor do atributo _val.
Inteiro operator()(int j)
{
    Inteiro i;
    i.val() = val()*j;
    return i;
}

}; // Fim da classe Inteiro.

int main(void)
{
    Inteiro i(2);

    // Sim, este código funciona.
    i = i(2)(3)(4);
    cout << i.val() << endl;

    return 0;
}

```



