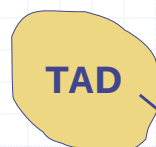


# Classes e Encapsulamento

*Marcio Santi*

POO e C++

## Tipo Abstrato de Dados (TAD)



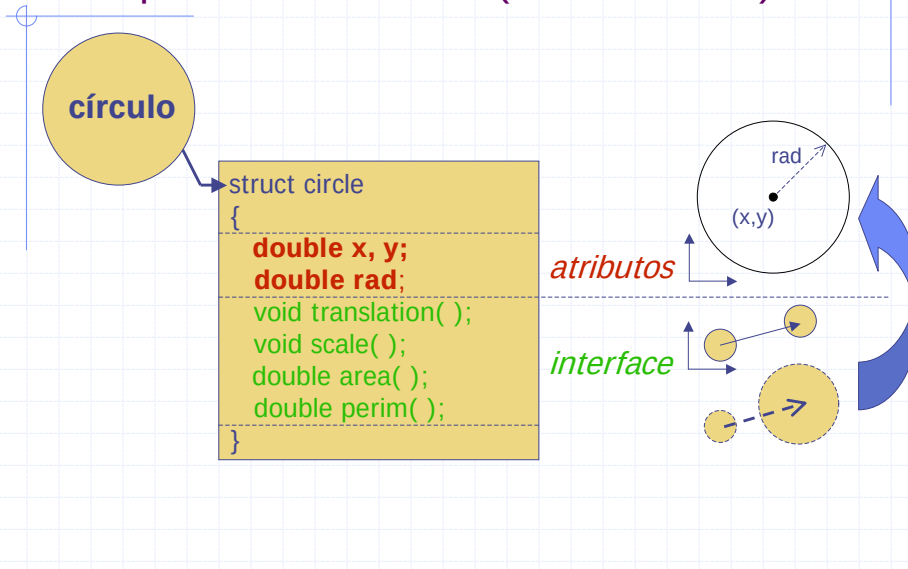
- **atributos**

- dados que o representam

- **interface**

- operações associadas
- como é manipulado

## Exemplo: Editor Gráfico (classe círculo)



## Conceito de classes em C++

- ◆ O conceito de classes em C++ pode ser visto como uma extensão à idéia de estruturas encontrada em C puro.
- ◆ É uma forma de se implementar um tipo e associar a ele funções e operadores.
- ◆ Uma classe em C++ é o elemento básico sobre o qual toda a orientação a objetos está apoiada.
- ◆ Classes (TAD) não são definidos pela sua representação interna, mas sim pelas operações sobre esse tipo (interface).
- ◆ Estas operações só fazem sentido quando associadas às suas representações.

## Funções como membros de estruturas

- ◆ Conceito de estruturas em C++ permite que funções sejam membros.
- ◆ Declaração da função é incluída na definição da estrutura.
- ◆ Define-se operações diretamente relacionadas a um TAD, exemplo:

```
stack data, operands;  
// ...  
data.reset( );  
operands.reset( );
```

- ◆ Função membro (ou método) reset atribui a data.top e a operands.top o valor EMPTY.

## Função como membros de estruturas

- ◆ Se declaramos um ponteiro para uma stack:

```
stack * ptr_operands = &operands;  
// ...  
ptr_operands -> push( 'A' );
```

- ◆ No trecho de código acima, a chamada do método push incrementa operands.top e atribui a operands.s[top] o valor 'A'

## Funções membro *inline*

- ◆ Funções membro, ou métodos, definidos dentro da classe são implicitamente *inline*.
- ◆ Para defini-las fora da estrutura, teríamos que utilizar o operador de escopo:

```
struct stack {  
    // ...  
    void push( char c ); // prototype ou assinatura do método  
};  
// ...  
void stack::push( char c )  
{  
    s[++top] = c;  
}
```

operador de escopo

## Sobrecarga de métodos

- ◆ É possível também sobrecarregar métodos em uma estrutura:

```
struct stack {  
    // ...  
    char pop( );  
    char pop( int n ); // sobrecarga à pop( )  
};  
char stack::pop( int n )  
{  
    while( (n-- > 1) && ( _top == 0 ) ) _top--;  
    return _s[_top--];  
}
```

## Acesso público e privado

- ◆ O conceito de estruturas em C++ também permite a definição das formas de acesso a seus dados.
- ◆ IDÉIA: esconder o máximo de informação possível.
- ◆ Para tal são impostas certas regras para acesso a atributos e métodos de uma classe.
- ◆ São estabelecidos três níveis de permissão de acordo com o contexto de uso dos atributos:
  - Privado: acesso restrito aos métodos da classe.
  - Público: acesso a partir de objetos da classe.
  - Protegido: acesso liberado a métodos de classes derivadas (depois).
- ◆ Cada um tem uma palavra reservada associada, *private*, *public* e *protected* respectivamente.

## Acesso público e privado

```
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.

## Membros de classes *private*

- ◆ Somente a própria classe pode acessar os atributos privados. Ou seja, somente os métodos da própria classe têm acesso a esses 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;  
}
```

## Membros de classes *public*

- ◆ É possível fazer chamadas e manipular respectivamente métodos e atributos da classe.
- ◆ Para poder chamar os métodos f1 e f2 no exemplo anterior é necessário defini-los como *public*:

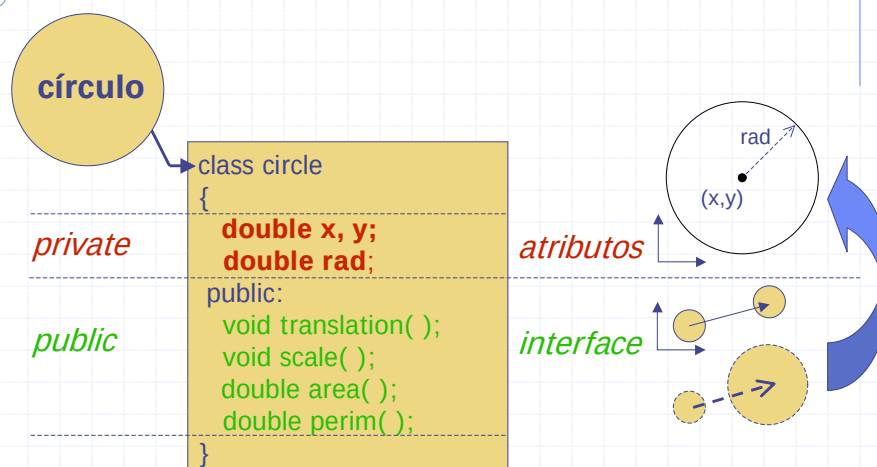
```
struct SemUso {  
    private:  
    int valor;  
    public:  
    void f1() { valor = 0; }  
    int f2() { return valor; }  
};  
  
// ...  
  
p.f1();                // ok, f1 é public!  
printf("%d", p.f2() ); // ok, f2 é public!
```

## Diferença entre *struct* e *class* em C++

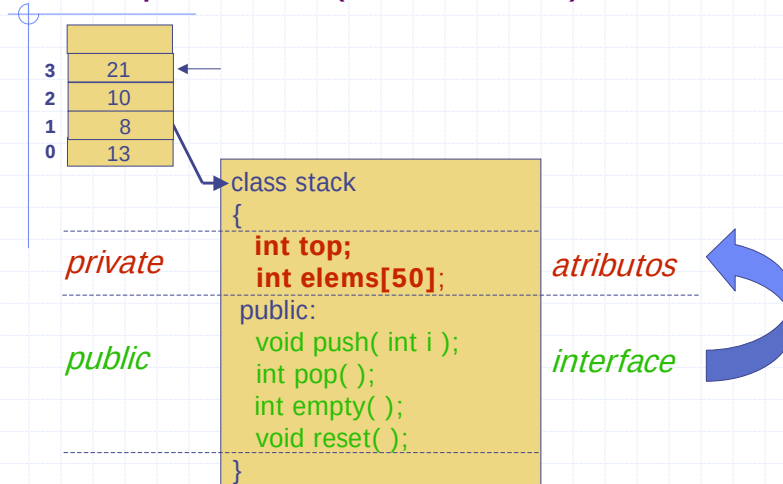
- ◆ Ambas podem representar, em C++, um TAD
- ◆ A diferença está no nível de proteção, caso nenhum dos especificadores de acesso seja usado:
  - *class*: por default é **private**
  - *struct*: por default é **public**

```
struct A {  
    int a; // a é público  
};  
  
class B {  
    int a; // a é privado  
};
```

## Exemplo: Editor Gráfico (classe círculo)



## Exemplo: Pilha (classe stack)



## Funções e classes *friend*

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

```
class point
{
    friend class circle;
private:
    double _x, _y;
public:
    point(double x, double y):_x(x),_y(y) { }
};

class circle
{
    double rad;
    point cnt;
public:
    circle(point c, double r): rad(r),cnt(c) { }
    void show( )
    {
        printf("x = %f\ny = %f\nr = %f\n", cnt._x, cnt._y, rad);
    }
};
```

## Funções e classes *friend*

- ◆ Em C++ funções também podem ser *friend* de uma classe:

```
class Stack {  
    friend Stack* createStack(void);  
    int top;  
    int elems[MAX];  
public:  
    void push(int i) { elems[++top]=i; }  
    int pop(void) { return elems[top--]; }  
    int empty(void) { return top == EMPTY; }  
};  
// ...  
Stack* createStack(void)  
{  
    Stack* s = new Stack;  
    s->top = EMPTY;  
    return s;  
}
```

## Funções e classes *friend*

- ◆ A idéia de métodos e classes *friend*, no entanto, quebra um dos conceitos básicos de POO, o encapsulamento.
- ◆ No exemplo da classe *stack* seria possível uma outra construção que evitaria o uso da palavra *friend*.

```
class Stack {  
    int top;  
    int elems[MAX];  
public:  
    void push(int i) { elems[++top]=i; }  
    int pop(void) { return elems[top--]; }  
    void reset( ) { top = EMPTY; }  
    int empty(void) { return top == EMPTY; }  
};
```

```
Stack* createStack(void)  
{  
    Stack* s = new Stack;  
    s->reset( );  
    return s;  
}
```

## Construtores e destrutores

- ◆ CONSTRUTOR: É uma função membro, ou método, usado para construir um objeto de uma dada classe e é chamado automaticamente quando um objeto é instanciado.
- ◆ DESTRUTOR: Analogamente, é um método chamado automaticamente quando um objeto é destruído.
- ◆ Os construtores são importantes para manter a consistência interna do objeto. O construtor deve garantir que o objeto recém criado seja consistente.
- ◆ No exemplo da classe *stack* era necessário antes de *usar* o objeto, ou seja, carregar a pilha, inicializá-lo através da função *reset()*.
- ◆ Mas algum cliente incauto da classe *stack* poderia inserir e eliminar elementos na pilha sem antes inicializá-la o que provocaria invasão de memória.
- ◆ O uso de construtores elimina essa possibilidade.

## Construtores: classe *stack*

```
class Stack {  
    pint top;  
    int elems[MAX];  
public:  
    void reset( )    { top = EMPTY; }  
    void push(int i) { elems[++top]=i; }  
    int pop(void)   { return elems[top--]; }  
    int empty(void) { return top == EMPTY; }  
};
```

```
int main( )  
{  
    // ...  
    Stack s;  
    s.reset( );  
    s.push(43);  
    // ...  
}
```

```
class Stack {  
    int top;  
    int elems[MAX];  
public:  
    Stack( )    { top = EMPTY; }  
    void push(int i) { elems[++top]=i; }  
    int pop(void)   { return elems[top--]; }  
    int empty(void) { return top == EMPTY; }  
};
```

```
int main( )  
{  
    // ...  
    Stack s;  
  
    s.push(43);  
    // ...  
}
```

## Construtores e destrutores: declaração

- ◆ Os construtores e destrutores são métodos especiais. Nenhum dos dois tem um tipo de retorno.
- ◆ O destrutor não pode receber parâmetros, ao contrário dos construtores.
- ◆ 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).
- ◆ Um construtor de uma classe **X** é declarado como um método de nome **X**, o nome do destrutor é **~X**.
- ◆ Pelo mecanismo de sobrecarga, classes podem ter mais de um construtor, ao contrário dos destrutores.

## Chamada de construtores e destrutores

- ◆ Os construtores são chamados automaticamente sempre que um objeto da classe for criado:
  - Quando a variável é declarada (alocado na pilha)
  - Quando o objeto é alocado com **new** (alocado do *heap*)
- ◆ Os destrutores de objetos alocados na pilha são chamados quando o objeto sai do seu escopo. Os destrutores de objetos criados com **new** são chamados com o operador **delete**.

```
class A {
public:
    A( ) { printf("construtor\n"); }
    ~A( ) { printf("destrutor\n"); }
};

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

## Exemplo: classe *stack*

```
class Stack {  
    int top;  
    int elems[MAX];  
public:  
    Stack( ) { top = EMPTY; }  
    void push(int i) { elems[++top]=i; }  
    int pop(void) { return elems[top--]; }  
    int empty(void) { return top == EMPTY; }  
};
```

```
int main( )  
{  
    Stack s;  
    s.push(43);  
    // ...  
}
```

```
int main( )  
{  
    Stack* s;  
    s = new Stack;  
    s->push(43);  
    // ...  
}
```

## Construtores com parâmetros

- ◆ No exemplo da classe *stack* o cliente da classe poderia querer definir um objeto pilha com uma dimensão maior do que 50:

```
class Stack {  
    int top;  
    int size;  
    char *s;  
public:  
    Stack( ) { top = EMPTY; size = max_len; s = new char[size]; }  
    Stack( int n ) // sobrecarga do construtor  
    {  
        top = EMPTY; size = n;  
        s = new char[size];  
    }  
    ~Stack( ) { delete [] s; }  
};
```

```
int main( )  
{  
    Stack s1;  
    Stack s2(2000);  
    Stack *s3 = new Stack;  
    Stack *s4 = new Stack(2000);  
}
```

## Construtores gerados automaticamente

- ◆ O fato de algumas classes não declararem construtores não significa que elas não tenham construtores. Na realidade o compilador gera dois construtores automaticamente, um *default* e um de cópia.
- ◆ Considere uma classe X:

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



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

```
void main()
{
    X a1; // usa construtor vazio
    X a2(a1); // usa construtor de cópia
    X a3 = a2; // usa construtor de cópia
}
```

## Construtor de cópia

- ◆ Dependendo dos tipos dos atributos de uma classe, é necessário implementar explicitamente um construtor de cópia. Exemplo:

```
class Stack {
    int top;
    char *s;
public:
    Stack( );
    Stack( int n );
    // ...
};
```

```
int main( )
```

```
{
    stack s1(50);
    // ...
    stack s2 = s1;
}
```

ERRO!

```
Stack::Stack( const Stack& t )
{
    this->size = t.size;
    this->top = t.top;
    s = new char[this->size];
    for(int i=0; i<top; ++i) this->s[i] = t.s[i];
}
```

## Métodos *const*

- ◆ Métodos *const* são métodos que não alteram o estado interno de um objeto. A especificação *const* faz parte da declaração do método:

```
class A {  
    public: int value;  
    public: int get ( )      { return value; }  
    public: 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  
}
```

- ◆ Para eliminar o erro seria necessário declarar o método *get* como sendo *const*:

```
class A {  
    // ...  
    public: int get const( )      { return value; }  
};
```

## This

- ◆ Em todo método não *static* (depois) 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.
- ◆ A palavra ***this*** é utilizada quando se necessita manipular o próprio objeto sobre o qual o método foi chamado.

## This - Exemplo

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

## Campos de classe *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*.
- ◆ A declaração apenas diz que a variável existe; algum módulo precisa defini-la.

## Campos *static* - Exemplo

```
class A {
public:
    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
}
```

- ◆ Como uma variável estática é única para todos os objetos da classe, não é necessário um objeto para referenciar este campo: `A::b = 4;`

## Métodos *static*

- ◆ Métodos *static* são como funções globais com escopo reduzido à classe.
- ◆ Isto significa que métodos *static* não apresentam o parâmetro *this*, ou seja, apenas os campos *static* podem ser acessados.

```
class A {
public:    int a;
public:    static int b;
public:    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
}
```