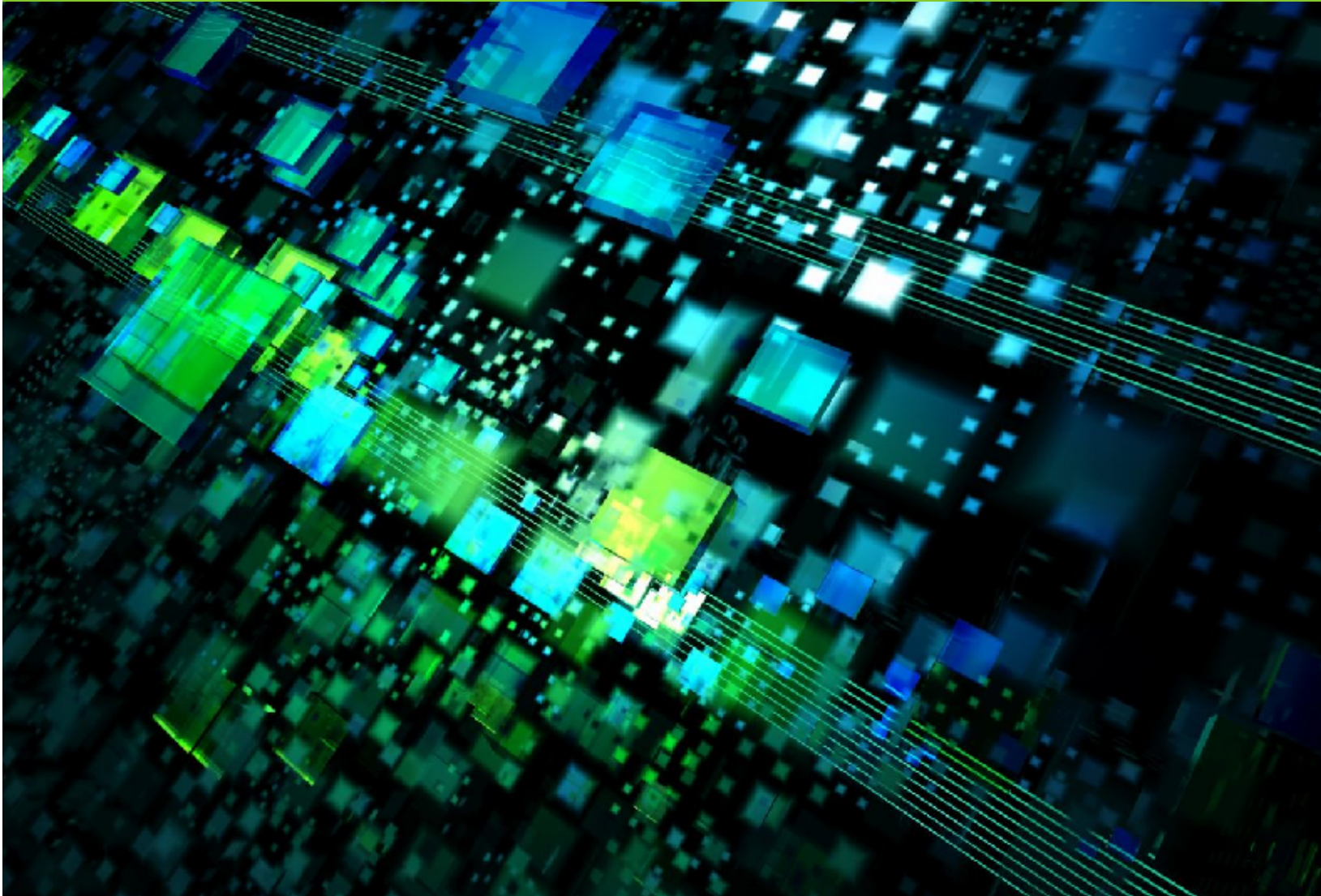


Programação de Computadores

Introdução a Orientação a Objetos

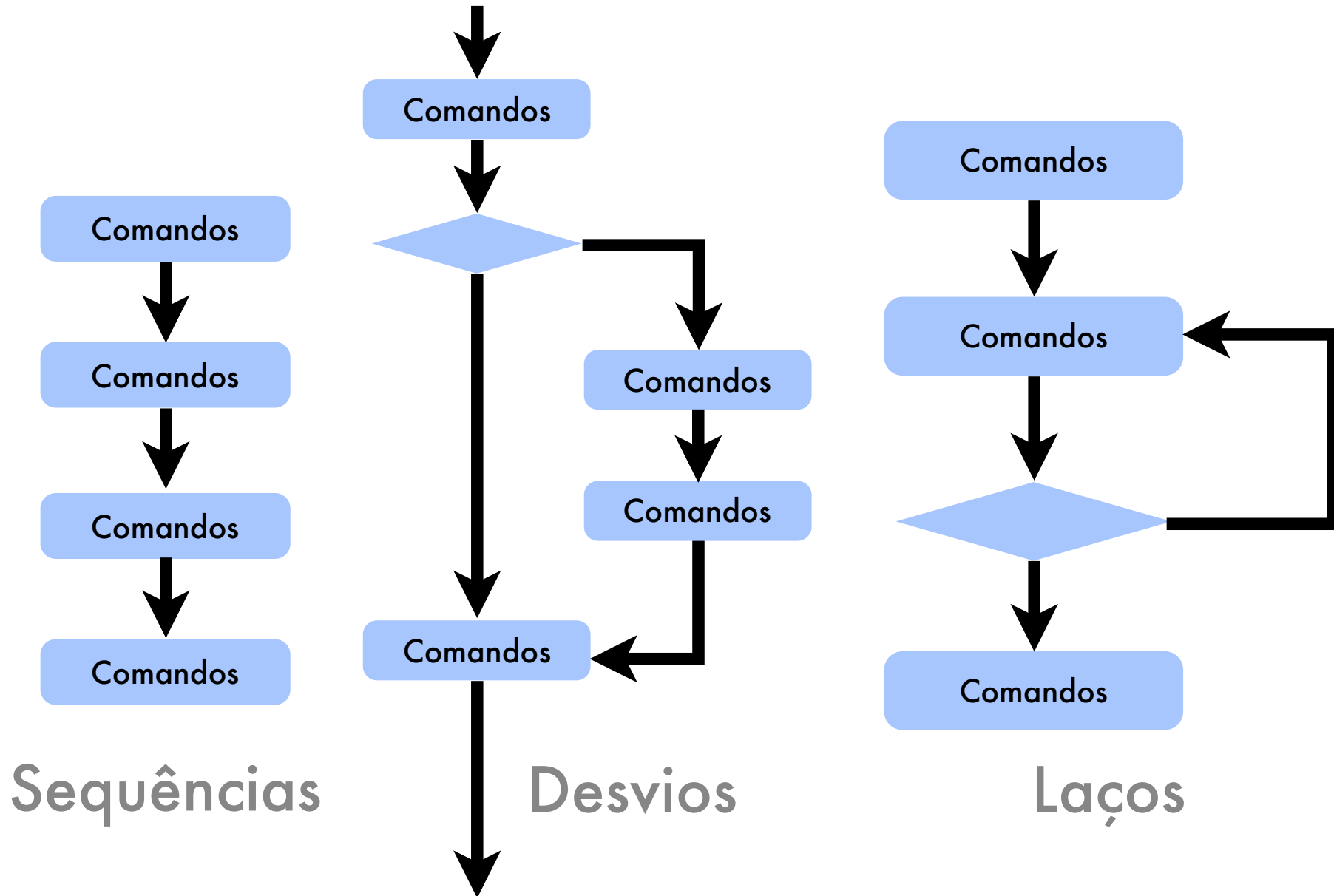


Alan
de
Freitas

Programação estruturada

- Em programação estruturada, o fluxo no qual ocorrem os comandos tem a maior importância.
- O foco está na ordem das ações.
- É assim que temos trabalhado até agora.

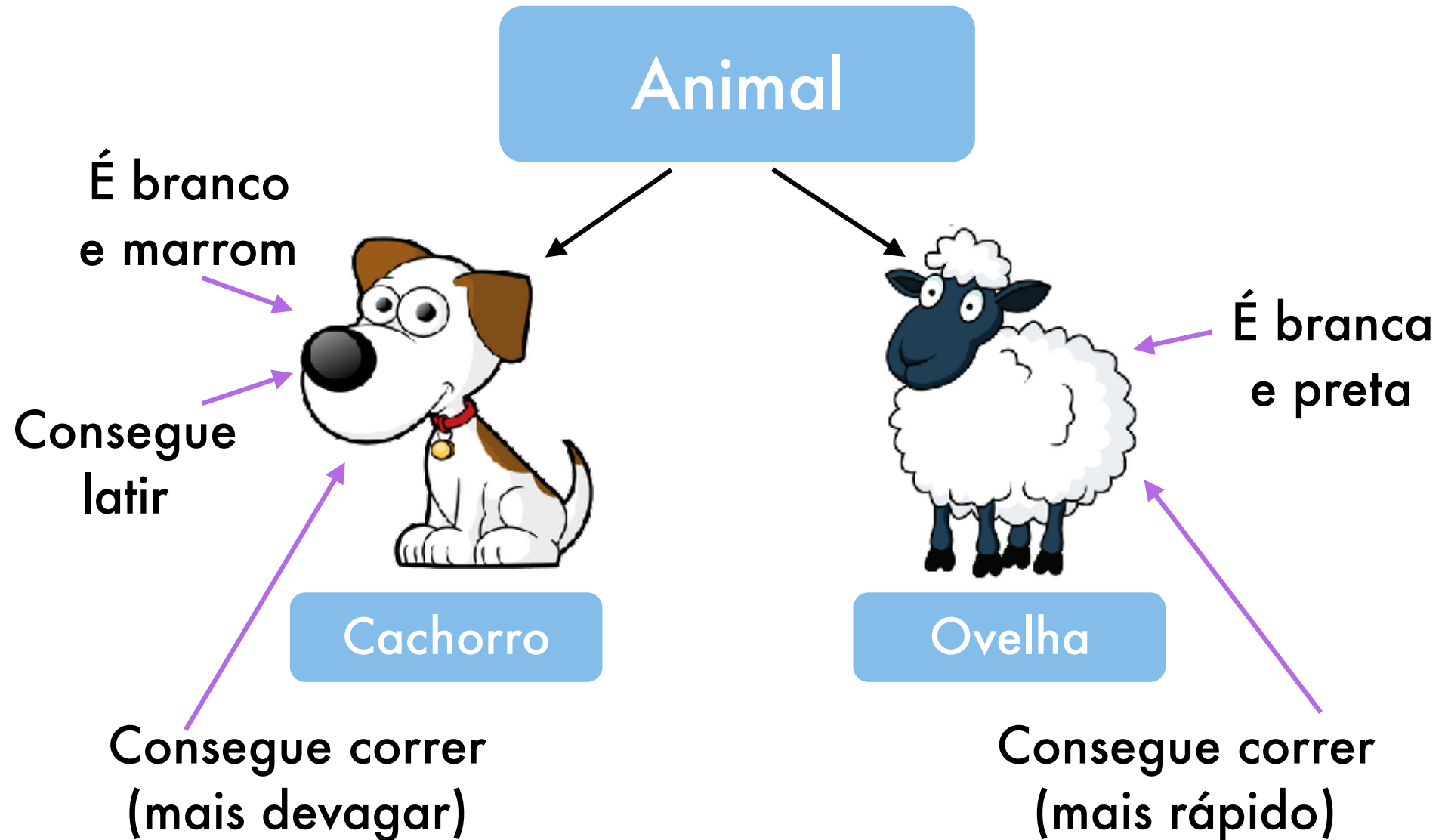
Programação estruturada



Programação Orientada a Objetos (POO)

- Em POO, a nossa preocupação é com os objetos.
- Dados os objetos, nos preocupamos em determinar como eles se comportam.
- O que cada objeto faz? Como é cada objeto?

Programação Orientada a Objetos (POO)



Struct e classes

- Para definir objetos novos, devemos criar novas classes.
- A relação entre classes e structs é bem direta.

Struct e classes

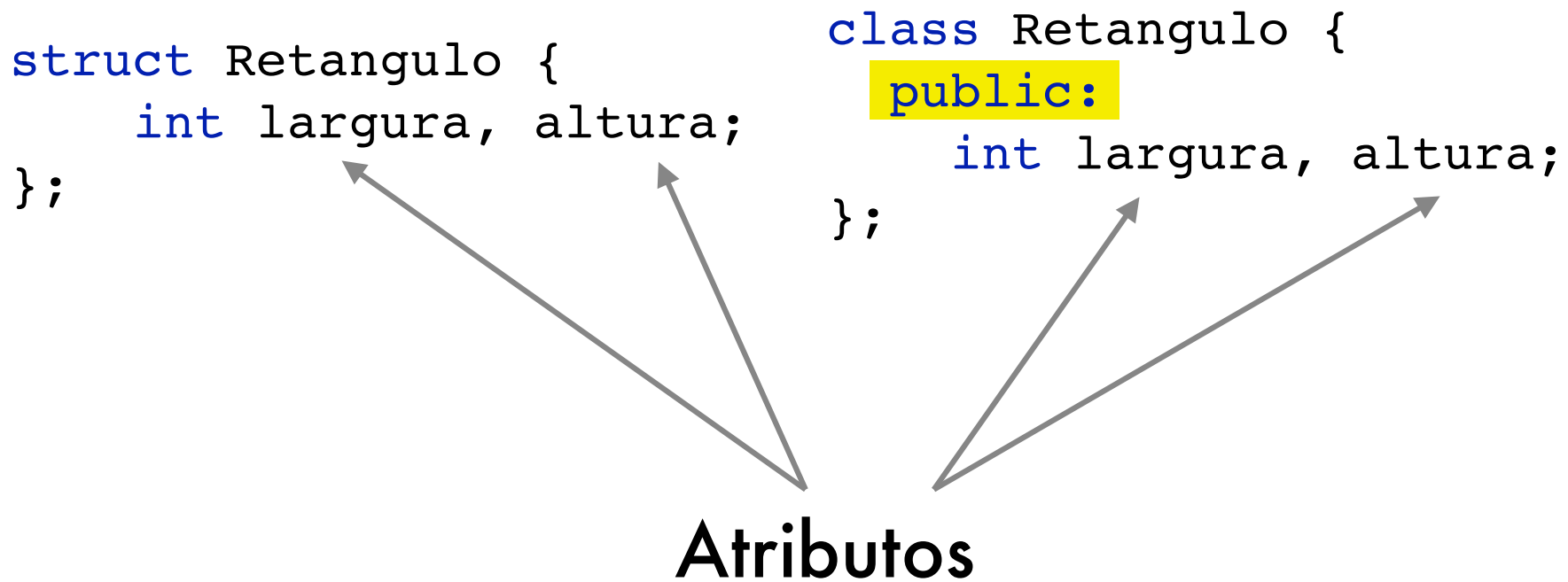
```
struct Retangulo {  
    int largura, altura;  
};
```

```
class Retangulo {  
    public:  
        int largura, altura;  
};
```

Atributos

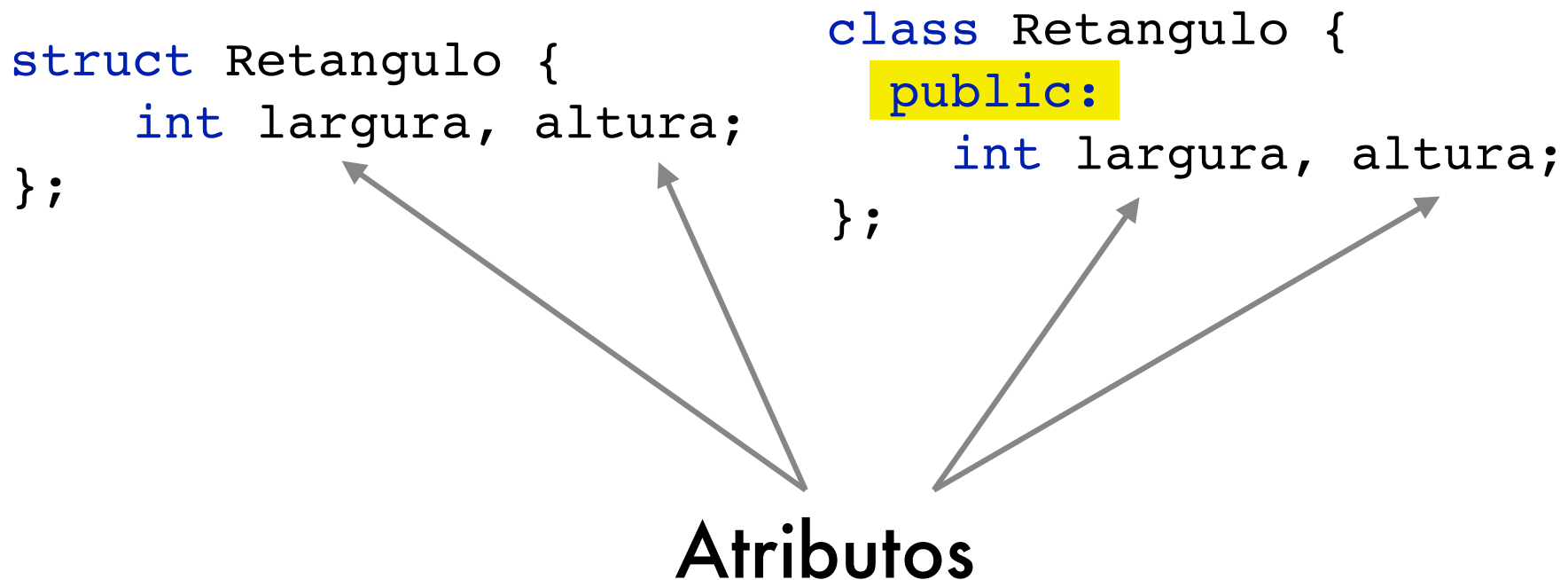
Veja o exemplo da criação de um novo tipo de dado que representa um retângulo.

Struct e classes



Já vimos como criar um novo tipo de dado com structs. Para fazer o mesmo com classes, devemos dizer que os membros da classe são públicos para que eles sejam acessíveis.

Struct e classes



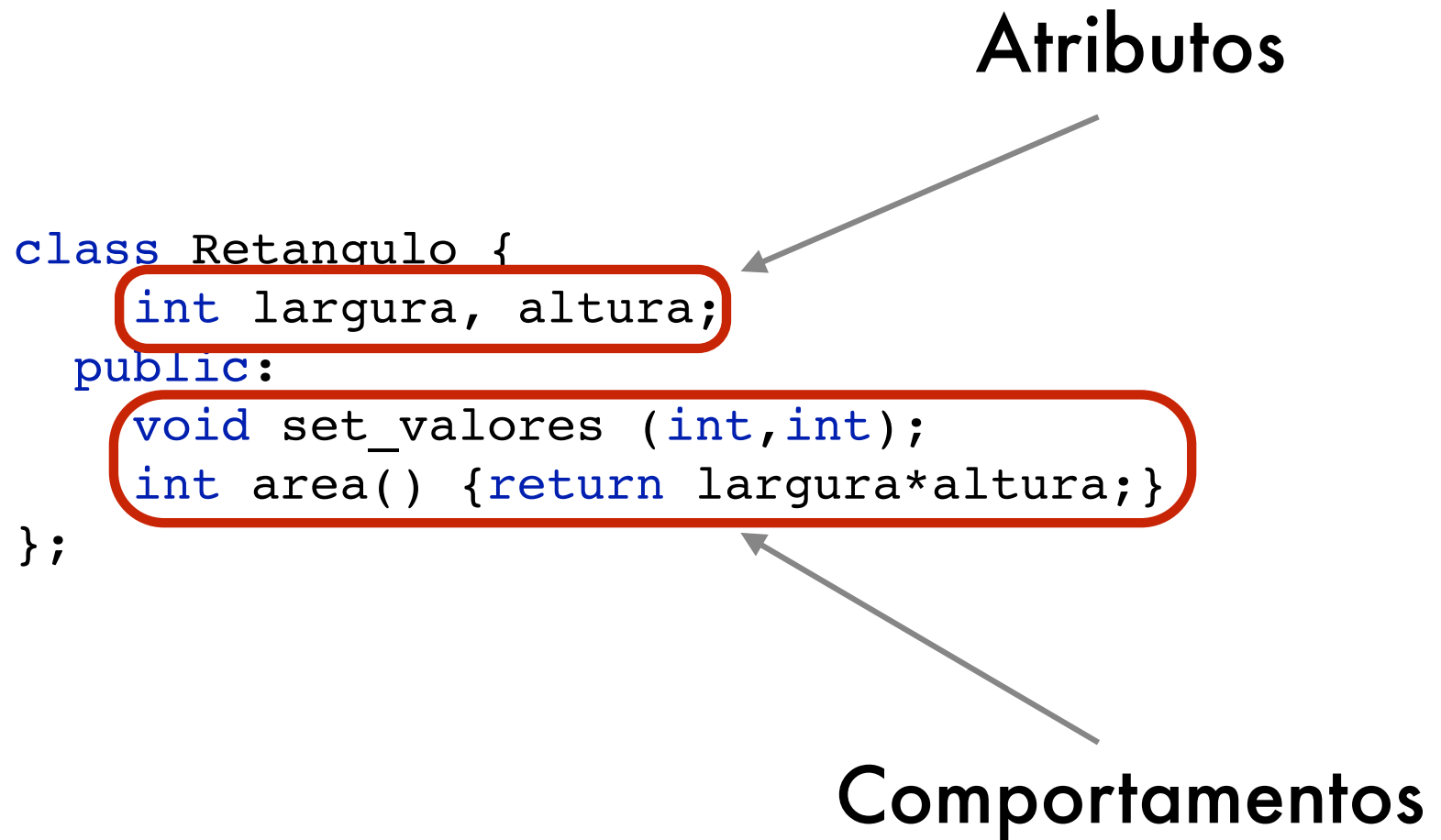
Sem a palavra chave `public`, os membros seriam considerados privados e não seriam acessíveis. Isto é um conceito importante em POO.

Funções como membros

```
class Retangulo {  
    int largura, altura;  
    public:  
        void set_valores (int,int);  
        int area() {return largura*altura;}  
};
```

Nos objetos, além de variáveis, podemos ter funções como membros de uma classe. As funções representam comportamentos de um objeto.

Funções como membros

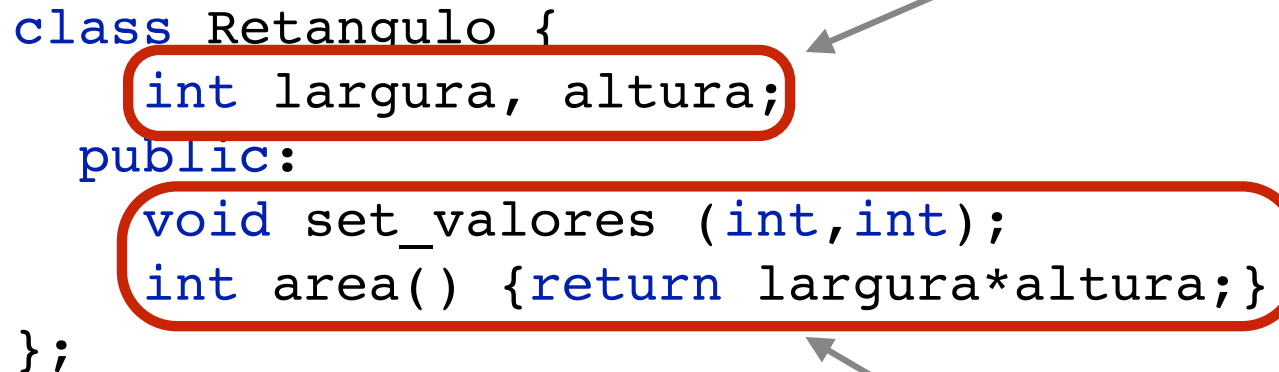


Funções como membros

Note que os atributos neste exemplo não são públicos.

Atributos privados

```
class Retangulo {  
    int largura, altura;  
public:  
    void set_valores (int, int);  
    int area() {return largura*altura;}  
};
```



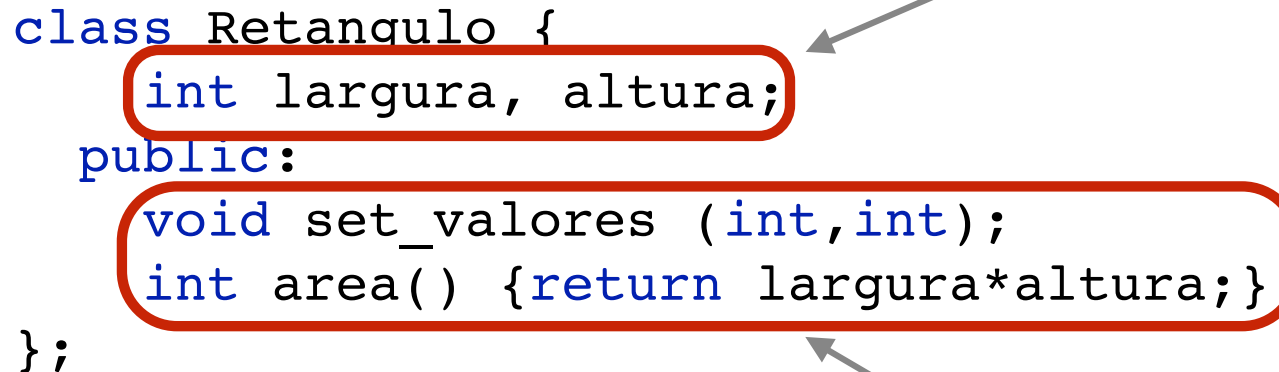
Comportamentos públicos

Funções como membros

Podemos ter em um objeto
membros públicos e
privados ao mesmo tempo

Membros privados

```
class Retangulo {  
    int largura, altura;  
public:  
    void set_valores (int, int);  
    int area() {return largura*altura;}  
};
```



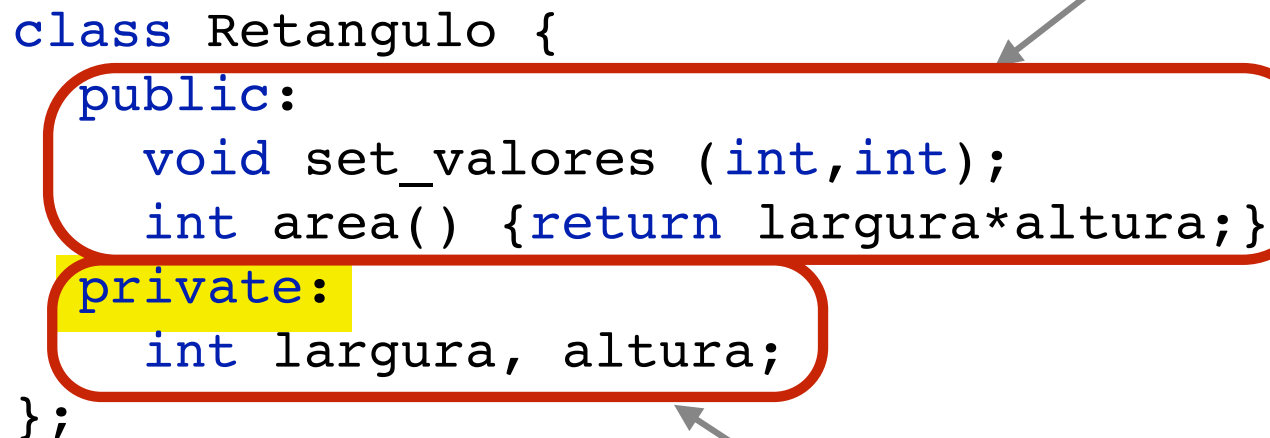
Membros públicos

Funções como membros

Na verdade, é comum deixar explícito quais são os membros privados.

Membros públicos

```
class Retangulo {  
    public:  
        void set_valores (int,int);  
        int area() {return largura*altura;}  
    private:  
        int largura, altura;  
};
```



Para isto usamos a palavra-chave `private`.

Membros privados

Funções como membros

```
class Retangulo {  
    public:  
        void set_valores (int,int);  
        int area() {return largura*altura;}  
    private:  
        int largura, altura;  
};
```

Como os atributos do retângulo não estão acessíveis.
Podemos utilizar a função pública `set_valores` para determinar os valores de `largura` e `altura`.

Funções como membros

```
class Retangulo {  
    public:  
        void set_valores (int,int);  
        int area() {return largura*altura;}  
    private:  
        int largura, altura;  
};
```

Isto é uma prática muito comum pois podemos usar a função para validar os valores inseridos para largura e altura, por exemplo, verificando se são maiores que zero.

Funções como membros

```
class Retangulo {  
    public:  
        void set_valores (int,int);  
        int area() {return largura*altura;}  
    private:  
        int largura, altura;  
};
```

Note também que apenas o cabeçalho da função está definido no objeto. A implementação da função deve vir depois.

Funções como membros

```
class Retangulo {  
    public:  
        void set_valores (int,int);  
        int area() {return largura*altura;}  
    private:  
        int largura, altura;  
};
```

Outra função definida é area. Esta função utiliza os atributos conhecidos para retornar a área do retângulo. Como esta é uma função simples, sua implementação já consta no objeto.

Funções como membros

```
class Retangulo {  
    public:  
        void set_valores (int,int);  
        int area() {return largura*altura;}  
    private:  
        int largura, altura;  
};
```

Note com a função `area`, como a orientação a objetos pode ser útil para agregar nos retângulos as funções que são associadas apenas a retângulos.

Funções como membros

```
class Retangulo {  
    public:  
        void set_valores(int, int);  
        int area () {return (largura*altura);}  
    private:  
        int largura, altura;  
};  
  
void Retangulo::set_valores (int x, int y) {  
    largura = x;  
    altura = y;  
}
```

No código acima, incluímos também a definição da função `set_valores`. A sintaxe `Retangulo::set_valores` é utilizada para especificar que a função pertence à classe `Retangulo`.

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    void set_valores(int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

void Retangulo::set_valores (int x, int y) {
    largura = x;
    altura = y;
}

int main(){
    Retangulo r;
    r.set_valores(3,4);
    cout << "A área de r é " << r.area() << endl;
    return 0;
}

```

Neste exemplo criamos um objeto Retângulo e imprimimos sua área.

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    void set_valores(int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

void Retangulo::set_valores (int x, int y) {
    largura = x;
    altura = y;
}

int main(){
    Retangulo r;
    r.set_valores(3,4);
    cout << "A área de r é " << r.area() << endl;
    return 0;
}

```

Definição da classe Retangulo. Como podem existir vários retângulos diferentes, definimos aqui uma classe de objetos.

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    void set_valores(int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

void Retangulo::set_valores (int x, int y) {
    largura = x;
    altura = y;
}

int main(){
    Retangulo r;
    r.set_valores(3,4);
    cout << "A área de r é " << r.area() << endl;
    return 0;
}

```

Criamos um retângulo r que ainda não tem seus atributos inicializados. Dizemos que o objeto r é uma instância da classe de objetos Retangulo.

```
#include <iostream>

using namespace std;

class Retangulo {
public:
    void set_valores(int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

void Retangulo::set_valores (int x, int y) {
    largura = x;
    altura = y;
}

int main(){
    Retangulo r;
    r.set_valores(3,4);
    cout << "A área de r é " << r.area() << endl;
    return 0;
}
```

A função `set_valores` do retângulo é utilizada para inicializar sua largura com 3 e altura com 4.

```
#include <iostream>

using namespace std;

class Retangulo {
public:
    void set_valores(int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

void Retangulo::set_valores (int x, int y) {
    largura = x;
    altura = y;
}

int main(){
    Retangulo r;
    r.set_valores(3,4);
    cout << "A área de r é " << r.area() << endl;
    return 0;
}
```

A própria função area do retângulo é utilizada para imprimir a área.

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    void set_valores(int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

void Retangulo::set_valores (int x, int y) {
    largura = x;
    altura = y;
}

int main(){
    Retangulo r;
    r.set_valores(3,4);
    cout << "A área de r é " << r.area() << endl;
    return 0;
}

```

Note que função não recebe parâmetros pois tudo que ela precisa é dos próprios atributos do retângulo.

```

#include <iostream>

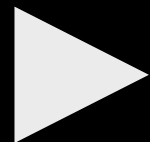
using namespace std;

class Retangulo {
public:
    void set_valores(int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

void Retangulo::set_valores (int x, int y) {
    largura = x;
    altura = y;
}

int main(){
    Retangulo r;
    r.set_valores(3,4);
    cout << "A área de r é " << r.area() << endl;
    return 0;
}

```



A área de r é 12

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    void set_valores(int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

void Retangulo::set_valores (int x, int y) {
    largura = x;
    altura = y;
}

int main(){
    Retangulo a,b;
    a.set_valores(3,4);
    b.set_valores(4,5);
    cout << "A área de a é " << a.area() << endl;
    cout << "A área de b é " << b.area() << endl;
    return 0;
}

```

Veja agora um exemplo com dois retângulos a e b.

Um objeto funciona independentemente do outro...

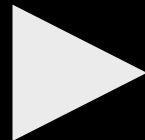
```
#include <iostream>

using namespace std;

class Retangulo {
public:
    void set_valores(int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

void Retangulo::set_valores (int x, int y) {
    largura = x;
    altura = y;
}

int main(){
    Retangulo a,b;
    a.set_valores(3,4);
    b.set_valores(4,5);
    cout << "A área de a é " << a.area() << endl;
    cout << "A área de b é " << b.area() << endl;
    return 0;
}
```



```
A área de a é 12
A área de b é 20
```

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    void set_valores(int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

void Retangulo::set_valores (int x, int y) {
    largura = x;
    altura = y;
}

int main(){
    Retangulo a;
    Retangulo *b = new Retangulo();
    a.set_valores(3,4);
    b->set_valores(4,5); // ou (*b).set_valores(4,5);
    cout << "A área de a é " << a.area() << endl;
    cout << "A área de *b é " << b->area() << endl;
    return 0;
}

```

Podemos também criar ponteiros para um objeto.

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    void set_valores(int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

void Retangulo::set_valores (int x, int y) {
    largura = x;
    altura = y;
}

int main(){
    Retangulo a;
    Retangulo *b = new Retangulo();
    a.set_valores(3,4);
    b->set_valores(4,5); // ou (*b).set_valores(4,5);
    cout << "A área de a é " << a.area() << endl;
    cout << "A área de *b é " << b->area() << endl;
    return 0;
}

```

Assim, como fazemos com structs, podemos utilizar o operador de ponteiro `->` em objetos.

```

#include <iostream>

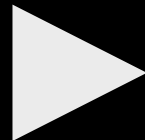
using namespace std;

class Retangulo {
public:
    void set_valores(int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

void Retangulo::set_valores (int x, int y) {
    largura = x;
    altura = y;
}

int main(){
    Retangulo a;
    Retangulo *b = new Retangulo();
    a.set_valores(3,4);
    b->set_valores(4,5); // ou (*b).set_valores(4,5);
    cout << "A área de a é " << a.area() << endl;
    cout << "A área de *b é " << b->area() << endl;
    return 0;
}

```



```

A área de a é 12
A área de *b é 20

```

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    void set_valores(int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

void Retangulo::set_valores (int x, int y) {
    largura = x;
    altura = y;
}

int main(){
    Retangulo a[2];
    a[0].set_valores(3,4);
    a[1].set_valores(4,5);
    cout << "A área de a[0] é " << a[0].area() << endl;
    cout << "A área de a[1] é " << a[1].area() << endl;
    return 0;
}

```

Como qualquer outro tipo de dado, podemos também criar arranjos de objetos.

```

#include <iostream>

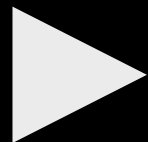
using namespace std;

class Retangulo {
public:
    void set_valores(int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

void Retangulo::set_valores (int x, int y) {
    largura = x;
    altura = y;
}

int main(){
    Retangulo a[2];
    a[0].set_valores(3,4);
    a[1].set_valores(4,5);
    cout << "A área de a[0] é " << a[0].area() << endl;
    cout << "A área de a[1] é " << a[1].area() << endl;
    return 0;
}

```



```

A área de a[0] é 12
A área de a[1] é 20

```

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    void set_valores(int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

void Retangulo::set_valores (int x, int y) {
    largura = x;
    altura = y;
}

int main(){
    Retangulo *a = new Retangulo[2];
    a[0].set_valores(3,4);
    a[1].set_valores(4,5);
    cout << "A área de a[0] é " << a[0].area() << endl;
    cout << "A área de a[1] é " << a[1].area() << endl;
    return 0;
}

```

Podemos também alocar arranjos de objetos dinamicamente.

```

#include <iostream>

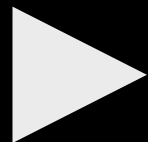
using namespace std;

class Retangulo {
public:
    void set_valores(int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

void Retangulo::set_valores (int x, int y) {
    largura = x;
    altura = y;
}

int main(){
    Retangulo *a = new Retangulo[2];
    a[0].set_valores(3,4);
    a[1].set_valores(4,5);
    cout << "A área de a[0] é " << a[0].area() << endl;
    cout << "A área de a[1] é " << a[1].area() << endl;
    return 0;
}

```



```

A área de a[0] é 12
A área de a[1] é 20

```

Programação Orientada a Objetos (POO)

- Como vimos, a POO serve para modelar sistemas mais complexos.
- Já que o mundo é rodeado de objetos, o conceito de objeto é útil para modelagem de coisas reais.
- Na POO, devemos modelar quais são os atributos (variáveis) e quais são os comportamentos (funções) de um objeto.

Construtores

- Quando um objeto é criado, uma função construtora é chamada.
- Esta função constrói o objeto dando valores iniciais a seus atributos.
- Para definir o construtor, criamos uma função com o mesmo nome da classe.

```
class Retangulo {  
    public:  
        Retangulo(int,int);  
        int area () {return (largura*altura);}  
    private:  
        int largura, altura;  
};
```

Nosso objeto tem agora uma função construtora que recebe dois números inteiros.

```
class Retangulo {  
    public:  
        Retangulo(int,int);  
        int area () {return (largura*altura);}  
    private:  
        int largura, altura;  
};  
  
Retangulo::Retangulo (int a, int b) {  
    largura = a;  
    altura = b;  
}
```

Na implementação da função construtora, guardamos o primeiro parâmetro como largura e o segundo como altura.

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    Retangulo(int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

int main () {
    Retangulo ra (3,4);
    Retangulo rb (5,6);
    cout << "A área de ra é " << ra.area() << endl;
    cout << "A área de rb é " << rb.area() << endl;
    return 0;
}

```

Ao criar um objeto do tipo Retangulo, os parâmetros para o construtor já devem ser declarados.

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    Retangulo(int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

int main () {
    Retangulo ra (3,4);
    Retangulo rb (5,6);
    cout << "A área de ra é " << ra.area() << endl;
    cout << "A área de rb é " << rb.area() << endl;
    return 0;
}

```

Como os objetos ra e rb já foram construídos com os valores de altura e largura, já podemos imprimir a área.

```

#include <iostream>

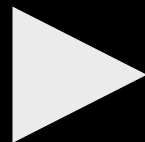
using namespace std;

class Retangulo {
public:
    Retangulo(int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

int main () {
    Retangulo ra (3,4);
    Retangulo rb (5,6);
    cout << "A área de ra é " << ra.area() << endl;
    cout << "A área de rb é " << rb.area() << endl;
    return 0;
}

```



```

Área de ra: 12
Área de rb: 30

```

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    Retangulo ();
    Retangulo (int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo () {
    largura = 5;
    altura = 5;
}

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

int main () {
    Retangulo ra(3,4);
    Retangulo rb;
    cout << "Área de ra: " << ra.area() << endl;
    cout << "Área de rb: " << rb.area() << endl;
    return 0;
}

```

Podemos também sobrecarregar construtores. Assim temos diferentes versões que dependem dos parâmetros.

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    Retangulo ();
    Retangulo (int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo () {
    largura = 5;
    altura = 5;
}

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

int main () {
    Retangulo ra(3,4);
    Retangulo rb;
    cout << "Área de ra: " << ra.area() << endl;
    cout << "Área de rb: " << rb.area() << endl;
    return 0;
}

```

No primeiro construtor, se nenhum parâmetro é passado, o Retangulo é criado com altura e largura 5.

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    Retangulo ();
    Retangulo (int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo () {
    largura = 5;
    altura = 5;
}

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

int main () {
    Retangulo ra(3,4);
    Retangulo rb;
    cout << "Área de ra: " << ra.area() << endl;
    cout << "Área de rb: " << rb.area() << endl;
    return 0;
}

```

No segundo construtor, se os parâmetros são passados, eles são atribuídos aos membros.

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    Retangulo ();
    Retangulo (int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo () {
    largura = 5;
    altura = 5;
}

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

int main () {
    Retangulo ra(3,4);
    Retangulo rb;
    cout << "Área de ra: " << ra.area() << endl;
    cout << "Área de rb: " << rb.area() << endl;
    return 0;
}

```

Criamos então neste exemplo um retângulo com cada construtor.

```

#include <iostream>

using namespace std;

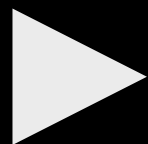
class Retangulo {
public:
    Retangulo ();
    Retangulo (int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo () {
    largura = 5;
    altura = 5;
}

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

int main () {
    Retangulo ra(3,4);
    Retangulo rb;
    cout << "Área de ra: " << ra.area() << endl;
    cout << "Área de rb: " << rb.area() << endl;
    return 0;
}

```



Área de ra: 12
 Área de rb: 25

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    Retangulo ();
    Retangulo (int);
    Retangulo (int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo () {
    largura = 5;
    altura = 5;
}

Retangulo::Retangulo (int a) {
    largura = a;
    altura = a;
}

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

int main () {
    Retangulo ra(3);
    Retangulo rb = 4;
    cout << "Área de ra:" << ra.area() << endl;
    cout << "Área de rb:" << rb.area() << endl;
    return 0;
}

```

Podemos criar quantas sobrecargas quisermos...

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    Retangulo ();
    Retangulo (int);
    Retangulo (int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo () {
    largura = 5;
    altura = 5;
}

Retangulo::Retangulo (int a) {
    largura = a;
    altura = a;
}

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

int main () {
    Retangulo ra(3);
    Retangulo rb = 4;
    cout << "Área de ra:" << ra.area() << endl;
    cout << "Área de rb:" << rb.area() << endl;
    return 0;
}

```

Neste exemplo, se apenas um parâmetro é passado, ele é atribuído tanto à altura quanto à largura.

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    Retangulo ();
    Retangulo (int);
    Retangulo (int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo () {
    largura = 5;
    altura = 5;
}

Retangulo::Retangulo (int a) {
    largura = a;
    altura = a;
}

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

int main () {
    Retangulo ra(3);
    Retangulo rb = 4;
    cout << "Área de ra:" << ra.area() << endl;
    cout << "Área de rb:" << rb.area() << endl;
    return 0;
}

```

Como vimos, o
parâmetros ou os
parâmetros devem ser
passados para o objeto
em sua criação.

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    Retangulo ();
    Retangulo (int);
    Retangulo (int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo () {
    largura = 5;
    altura = 5;
}

Retangulo::Retangulo (int a) {
    largura = a;
    altura = a;
}

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

int main () {
    Retangulo ra(3);
    Retangulo rb = 4;
    cout << "Área de ra:" << ra.area() << endl;
    cout << "Área de rb:" << rb.area() << endl;
    return 0;
}

```

Construtores com apenas um parâmetro são interessantes pois podem ser utilizados para atribuir um tipo de dado a outro. Neste exemplo, atribuímos um int em Retangulo.

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    Retangulo ();
    Retangulo (int);
    Retangulo (int,int);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

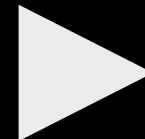
Retangulo::Retangulo () {
    largura = 5;
    altura = 5;
}

Retangulo::Retangulo (int a) {
    largura = a;
    altura = a;
}

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

int main () {
    Retangulo ra(3);
    Retangulo rb = 4;
    cout << "Área de ra:" << ra.area() << endl;
    cout << "Área de rb:" << rb.area() << endl;
    return 0;
}

```



Área de ra: 9
 Área de rb: 16

Outro tipo comum de construtor é o construtor de cópia.

```
#include <iostream>

using namespace std;

class Retangulo {
public:
    Retangulo (int,int);
    Retangulo (const Retangulo &);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

Retangulo::Retangulo (const Retangulo &direita) {
    largura = direita.largura;
    altura = direita.altura;
}

int main () {
    Retangulo ra(3,4);
    Retangulo rb = ra;
    cout << "Área de ra:" << ra.area() << endl;
    cout << "Área de rb:" << rb.area() << endl;
    return 0;
}
```

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    Retangulo (int,int);
    Retangulo (const Retangulo &);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

Retangulo::Retangulo (const Retangulo &direita) {
    largura = direita.largura;
    altura = direita.altura;
}

int main () {
    Retangulo ra(3,4);
    Retangulo rb = ra;
    cout << "Área de ra:" << ra.area() << endl;
    cout << "Área de rb:" << rb.area() << endl;
    return 0;
}

```

Este construtor recebe outro elemento do mesmo tipo e faz uma cópia.

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    Retangulo (int,int);
    Retangulo (const Retangulo &);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

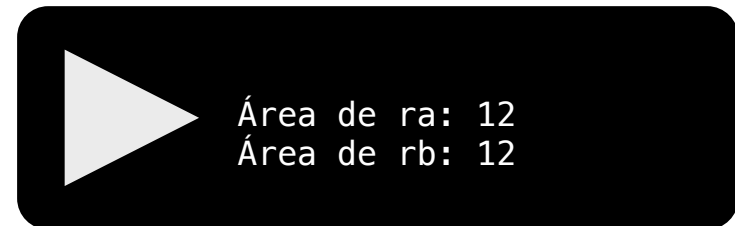
Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

Retangulo::Retangulo (const Retangulo &direita) {
    largura = direita.largura;
    altura = direita.altura;
}

int main () {
    Retangulo ra(3,4);
    Retangulo rb = ra;
    cout << "Área de ra:" << ra.area() << endl;
    cout << "Área de rb:" << rb.area() << endl;
    return 0;
}

```

No exemplo agora cria um Retangulo ra e uma cópia dele em rb.



Sobrecarga de operadores

- Imagine ter também a possibilidade de somar, dividir ou comparar dois objetos
- A sobrecarga de operadores define o que fazer quando são chamados os operadores para uma classe
- Os operadores que podem ser sobrecarregados são:

+	-	*	/	=	<	>	+=	-=	*=	/=	<<	>>
<<=	>>=	==	!=	<=	>=	++	--	%	&	^	!	
~	&=	^=	=	&&		%=	[]	()	,	->*	->	new
delete		new[]		delete[]								

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    Retangulo(int,int);
    Retangulo operator + (const Retangulo &);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

Retangulo Retangulo::operator+ (const Retangulo& direita) {
    Retangulo temp(largura + direita.largura, altura + direita.altura);
    return temp;
}

int main () {
    Retangulo ra(3,4);
    Retangulo rb(4,5);
    Retangulo rc = ra + rb;
    cout << "Área de rc: " << rc.area() << endl;
    return 0;
}

```

Neste exemplo criamos uma função nos permitirá somar dois retângulos.

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    Retangulo(int,int);
    Retangulo operator + (const Retangulo &);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

Retangulo Retangulo::operator+ (const Retangulo& direita) {
    Retangulo temp(largura + direita.largura, altura + direita.altura);
    return temp;
}

int main () {
    Retangulo ra(3,4);
    Retangulo rb(4,5);
    Retangulo rc = ra + rb;
    cout << "Área de rc: " << rc.area() << endl;
    return 0;
}

```

Para definir a função de soma usaremos como nome da função o operador +

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    Retangulo(int,int);
    Retangulo operator + (const Retangulo &);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

Retangulo Retangulo::operator+ (const Retangulo& direita) {
    Retangulo temp(largura + direita.largura, altura + direita.altura);
    return temp;
}

int main () {
    Retangulo ra(3,4);
    Retangulo rb(4,5);
    Retangulo rc = ra + rb;
    cout << "Área de rc: " << rc.area() << endl;
    return 0;
}

```

Esta operação de soma receberá um outro Retângulo ao qual a instância da classe será somada.

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    Retangulo(int,int);
    Retangulo operator + (const Retangulo &);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

Retangulo Retangulo::operator+ (const Retangulo& direita) {
    Retangulo temp(largura + direita.largura, altura + direita.altura);
    return temp;
}

int main () {
    Retangulo ra(3,4);
    Retangulo rb(4,5);
    Retangulo rc = ra + rb;
    cout << "Área de rc: " << rc.area() << endl;
    return 0;
}

```

O resultado da soma será também um outro retângulo.

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    Retangulo(int,int);
    Retangulo operator + (const Retangulo &);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

Retangulo Retangulo::operator+ (const Retangulo& direita) {
    Retangulo temp(largura + direita.largura, altura + direita.altura);
    return temp;
}

int main () {
    Retangulo ra(3,4);
    Retangulo rb(4,5);
    Retangulo rc = ra + rb;
    cout << "Área de rc: " << rc.area() << endl;
    return 0;
}

```

Na implementação da função, um novo Retângulo temp construído com a soma das larguras e alturas é construído.

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    Retangulo(int,int);
    Retangulo operator + (const Retangulo &);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

Retangulo Retangulo::operator+ (const Retangulo& direita) {
    Retangulo temp(largura + direita.largura, altura + direita.altura);
    return temp;
}

int main () {
    Retangulo ra(3,4);
    Retangulo rb(4,5);
    Retangulo rc = ra + rb;
    cout << "Área de rc: " << rc.area() << endl;
    return 0;
}

```

Este retângulo temp é retornado como resultado da soma de dois retângulos.

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    Retangulo(int,int);
    Retangulo operator + (const Retangulo &);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

Retangulo Retangulo::operator+ (const Retangulo& direita) {
    Retangulo temp(largura + direita.largura, altura + direita.altura);
    return temp;
}

int main () {
    Retangulo ra(3,4);
    Retangulo rb(4,5);
    Retangulo rc = ra + rb;
    cout << "Área de rc: " << rc.area() << endl;
    return 0;
}

```

Na função principal, usamos estes recursos para criar um retângulo rc como a soma de ra e rb

```

#include <iostream>

using namespace std;

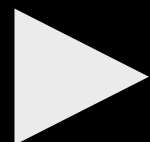
class Retangulo {
public:
    Retangulo(int,int);
    Retangulo operator + (const Retangulo &);
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

Retangulo Retangulo::operator+ (const Retangulo& direita) {
    Retangulo temp(largura + direita.largura, altura + direita.altura);
    return temp;
}

int main () {
    Retangulo ra(3,4);
    Retangulo rb(4,5);
    Retangulo rc = ra + rb;
    cout << "Área de rc: " << rc.area() << endl;
    return 0;
}

```



Área de rc: 63

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    Retangulo(int,int);
    bool operator < (Retangulo &dir) {return area() < dir.area();}
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

int main () {
    Retangulo ra(3,4);
    Retangulo rb(5,3);
    if (ra < rb){
        cout << "A área de ra é menor que a de rb" << endl;
    } else {
        cout << "A área de rb é menor que a de ra" << endl;
    }
    return 0;
}

```

Neste exemplo usamos o recurso de sobrecarga de operadores para podermos comparar dois retângulos de acordo com suas áreas.

```

#include <iostream>

using namespace std;

class Retangulo {
public:
    Retangulo(int,int);
    bool operator < (Retangulo &dir) {return area() < dir.area();}
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

int main () {
    Retangulo ra(3,4);
    Retangulo rb(5,3);
    if (ra < rb){
        cout << "A área de ra é menor que a de rb" << endl;
    } else {
        cout << "A área de rb é menor que a de ra" << endl;
    }
    return 0;
}

```

○ operador < retorna um bool de acordo com a área de cada retângulo.

```

#include <iostream>

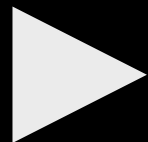
using namespace std;

class Retangulo {
public:
    Retangulo(int,int);
    bool operator < (Retangulo &dir) {return area() < dir.area();}
    int area () {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo (int a, int b) {
    largura = a;
    altura = b;
}

int main () {
    Retangulo ra(3,4);
    Retangulo rb(5,3);
    if (ra < rb){
        cout << "A área de ra é menor que a de rb" << endl;
    } else {
        cout << "A área de rb é menor que a de ra" << endl;
    }
    return 0;
}

```



A área de ra é menor que a de rb

Setters e Getters

- Funções que atribuem ou retornam atributos de um objeto.
- Deixam o objeto menos propenso a erros do que se deixássemos os atributos como públicos.

Na nova classe temos
2 setters e 2 getters
que servem para
enviar e receber
valores relativos às
variáveis.

```
class Retangulo {  
    public:  
        Retangulo(int,int);  
        void setAltura(int x);  
        void setLargura(int x);  
        int getAltura() {return altura;}  
        int getLargura() {return largura;}  
        int area() {return (largura*altura);}  
    private:  
        int largura, altura;  
};  
  
Retangulo::Retangulo(int a, int b) {  
    setAltura(a);  
    setLargura(b);  
}  
  
void Retangulo::setAltura(int x) {  
    if (x >=0){  
        altura = x;  
    } else {  
        altura = 0;  
    }  
}  
  
void Retangulo::setLargura(int x) {  
    if (x >=0){  
        largura = x;  
    } else {  
        largura = 0;  
    }  
}
```

```
class Retangulo {  
    public:  
        Retangulo(int,int);  
        void setAltura(int x);  
        void setLargura(int x);  
        int getAltura() {return altura;}  
        int getLargura() {return largura;}  
        int area() {return (largura*altura);}  
    private:  
        int largura, altura;  
};
```

```
Retangulo::Retangulo(int a, int b) {  
    setAltura(a);  
    setLargura(b);  
}
```

```
void Retangulo::setAltura(int x) {  
    if (x >=0){  
        altura = x;  
    } else {  
        altura = 0;  
    }  
}  
  
void Retangulo::setLargura(int x) {  
    if (x >=0){  
        largura = x;  
    } else {  
        largura = 0;  
    }  
}
```

As funções que dão valores aos atributos conferem se os valores são maiores do que zero pois a altura e largura de um retângulo não podem ser valores negativos.

```

class Retangulo {
public:
    Retangulo(int,int);
    void setAltura(int x);
    void setLargura(int x);
    int getAltura() {return altura;}
    int getLargura() {return largura;}
    int area() {return (largura*altura);}
private:
    int largura, altura;
};

```

```

Retangulo::Retangulo(int a, int b) {
    setAltura(a);
    setLargura(b);
}

```

```

void Retangulo::setAltura(int x) {
    if (x >=0){
        altura = x;
    } else {
        altura = 0;
    }
}

```

```

void Retangulo::setLargura(int x) {
    if (x >=0){
        largura = x;
    } else {
        largura = 0;
    }
}

```

Já a função construtora que define os valores iniciais de altura e largura utiliza os próprios setters para garantir que os valores serão válidos.

Na função principal,
tentamos atribuir valores
inválidos para um retângulo
mas a função não nos
permite.

```
#include <iostream>

using namespace std;

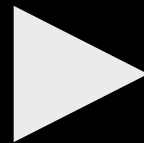
class Retangulo {
public:
    Retangulo(int,int);
    void setAltura(int x);
    void setLargura(int x);
    int getAltura() {return altura;}
    int getLargura() {return largura;}
    int area() {return (largura*altura);}
private:
    int largura, altura;
};

Retangulo::Retangulo(int a, int b) {
    setAltura(a);
    setLargura(b);
}

void Retangulo::setAltura(int x) {
    if (x >=0){
        altura = x;
    } else {
        altura = 0;
    }
}

void Retangulo::setLargura(int x) {
    if (x >=0){
        largura = x;
    } else {
        largura = 0;
    }
}
```

```
int main () {
    Retangulo ra(3,2);
    cout << "A área de ra é " << ra.area() << endl;
    ra.setAltura(4);
    ra.setLargura(7);
    cout << "A área de ra é " << ra.area() << endl;
    ra.setAltura(3);
    ra.setLargura(-5); // tentando passar um valor inválido
    cout << "A área de ra é " << ra.area() << endl;
    return 0;
}
```



A área de ra é 6
A área de ra é 28
A área de ra é 0

Setters and Getters

- A utilização deste recurso tem várias vantagens:
 - Se for necessário alterar várias variáveis no objeto, o usuário da classe não precisa fazer isto
 - Os dados podem ser validados
 - O código do usuário continua válido se você quiser alterar como o setter funcionará
 - O modo como o objeto está sendo representado fica oculto

Destrutores

- Além dos construtores, podemos definir quais comandos serão executados quando um objeto for destruído
- Pode parecer estranho, mas isto é especialmente importante para evitar vazamento de memória quando destruirmos objetos que haviam alocado espaço na memória

```
class Lista {  
public:  
    Lista(int);  
    void setElemento(int pos, int x) {px[pos-1] = x;}  
    int getElemento(int pos) {return px[pos-1];}  
private:  
    int *px;  
    int tamanho;  
};  
  
Lista::Lista(int n){  
    tamanho = n;  
    px = new int[n];  
    for (int i = 0; i < n; i++){  
        px[i] = i+1;  
    }  
}
```

Suponha uma classe utilizada para representar listas de números inteiros. Como não sabemos qual será o tamanho da lista, utilizaremos alocação dinâmica de memória.

```

class Lista {
public:
    Lista(int);
    void setElemento(int pos, int x) {px[pos-1] = x;}
    int getElemento(int pos) {return px[pos-1];}
private:
    int *px;
    int tamanho;
};

Lista::Lista(int n){
    tamanho = n;
    px = new int[n];
    for (int i = 0; i < n; i++){
        px[i] = i+1;
    }
}

```

Um ponteiro `px` apontará para o arranjo alocado na memória e um outro membro guardará o tamanho do arranjo alocado, que é também o tamanho da lista.

```

class Lista {
public:
    Lista(int);
    void setElemento(int pos, int x) {px[pos-1] = x;}
    int getElemento(int pos) {return px[pos-1];}
private:
    int *px;
    int tamanho;
};

Lista::Lista(int n){
    tamanho = n;
    px = new int[n];
    for (int i = 0; i < n; i++){
        px[i] = i+1;
    }
}

```

○ construtor recebe o tamanho da lista de elementos.

```

class Lista {
public:
    Lista(int);
    void setElemento(int pos, int x) {px[pos-1] = x;}
    int getElemento(int pos) {return px[pos-1];}
private:
    int *px;
    int tamanho;
};

Lista::Lista(int n){
    tamanho = n;
    px = new int[n];
    for (int i = 0; i < n; i++){
        px[i] = i+1;
    }
}

```

Em sua implementação, o construtor aloca um arranjo do tamanho pedido e dá valores de 1 a n para os elementos da lista.

```

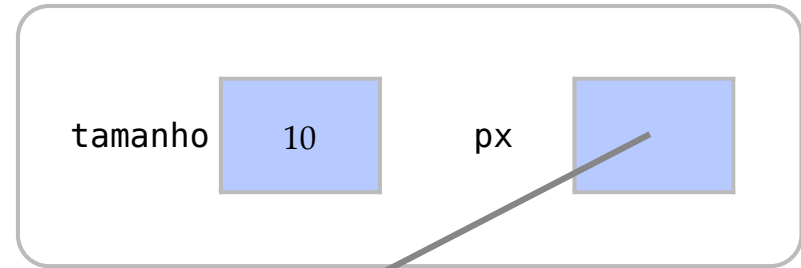
class Lista {
public:
    Lista(int);
    void setElemento(int pos, int x) {px[pos-1] = x;}
    int getElemento(int pos) {return px[pos-1];}
private:
    int *px;
    int tamanho;
};

Lista::Lista(int n){
    tamanho = n;
    px = new int[n];
    for (int i = 0; i < n; i++){
        px[i] = i+1;
    }
}

```

Os setters e getters retornam o elemento em uma posição da lista. A função considera que as posições na lista são de 1 até n.

Atributos da lista

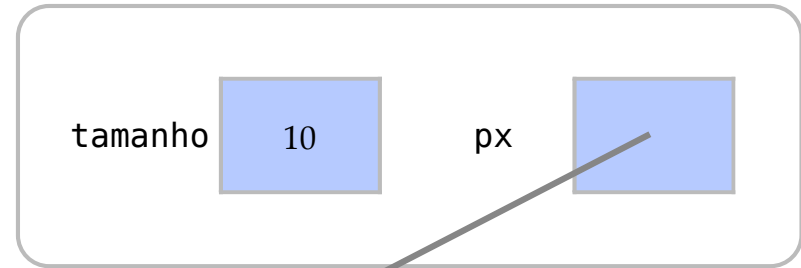


```
class Lista {  
    public:  
        Lista(int);  
        void setElemento(int pos, int x) {px[pos-1] = x;}  
        int getElemento(int pos) {return px[pos-1];}  
    private:  
        int *px;  
        int tamanho;  
};  
  
Lista::Lista(int n){  
    tamanho = n;  
    px = new int[n];  
    for (int i = 0; i < n; i++){  
        px[i] = i+1;  
    }  
}
```



Veja agora o que acontece em relação aos atributos do objeto quando ele é construído.

Atributos da lista

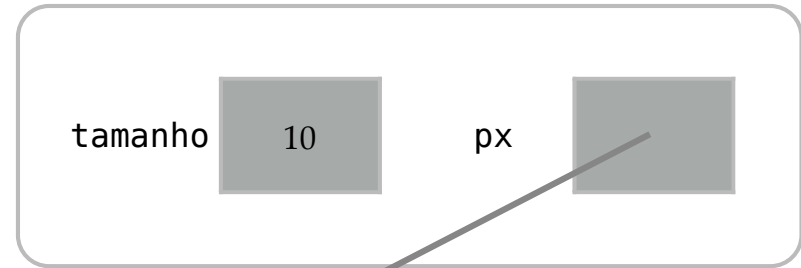


```
class Lista {  
    public:  
        Lista(int);  
        void setElemento(int pos, int x) {px[pos-1] = x;}  
        int getElemento(int pos) {return px[pos-1];}  
    private:  
        int *px;  
        int tamanho;  
};  
  
Lista::Lista(int n){  
    tamanho = n;  
    px = new int[n];  
    for (int i = 0; i < n; i++){  
        px[i] = i+1;  
    }  
}
```



Apenas o ponteiro e o tamanho são atributos da lista. O arranjo, apesar de ser usado para representar a lista, está alocado na memória.

Atributos da lista

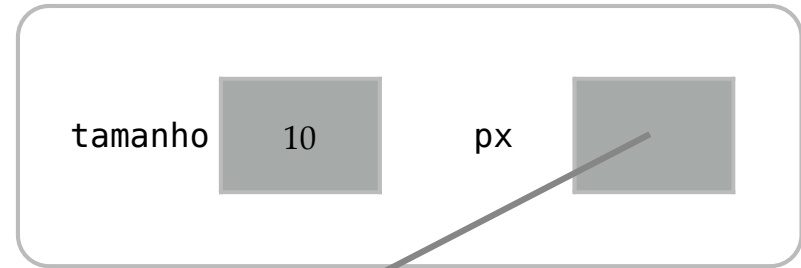


```
class Lista {  
    public:  
        Lista(int);  
        void setElemento(int pos, int x) {px[pos-1] = x;}  
        int getElemento(int pos) {return px[pos-1];}  
    private:  
        int *px;  
        int tamanho;  
};  
  
Lista::Lista(int n){  
    tamanho = n;  
    px = new int[n];  
    for (int i = 0; i < n; i++){  
        px[i] = i+1;  
    }  
}
```



Quando esta lista é destruída, apenas o ponteiro é destruído. O arranjo continua existindo, o que causa um vazamento de memória.

Atributos da lista



```
class Lista {  
    public:  
        Lista(int);  
        void setElemento(int pos, int x) {px[pos-1] = x;}  
        int getElemento(int pos) {return px[pos-1];}  
    private:  
        int *px;  
        int tamanho;  
};  
  
Lista::Lista(int n){  
    tamanho = n;  
    px = new int[n];  
    for (int i = 0; i < n; i++){  
        px[i] = i+1;  
    }  
}
```



Para que isto não ocorra, precisamos definir na função destrutora que o arranjo será desalocado.

```

class Lista {
public:
    Lista(int);
    ~Lista();
    void setElemento(int pos, int x) {px[pos-1] = x;}
    int getElemento(int pos) {return px[pos-1];}
private:
    int *px;
    int tamanho;
};

Lista::Lista(int n){
    tamanho = n;
    px = new int[n];
    for (int i = 0; i < n; i++){
        px[i] = i+1;
    }
}

Lista::~Lista(){
    delete[] px;
}

```

Nesta nova definição da classe, temos uma função destrutura. A função destrutura tem o mesmo nome da classe precedido de um til ~.

```

class Lista {
public:
    Lista(int);
    ~Lista();
    void setElemento(int pos, int x) {px[pos-1] = x;}
    int getElemento(int pos) {return px[pos-1];}
private:
    int *px;
    int tamanho;
};

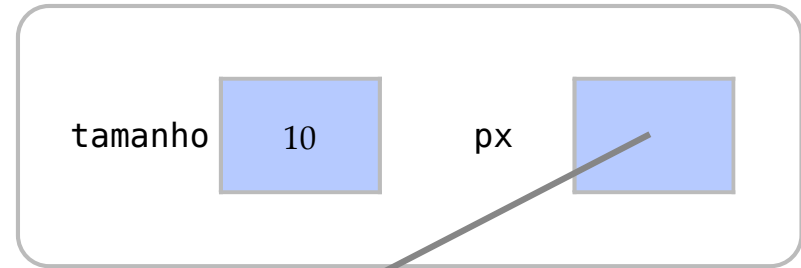
Lista::Lista(int n){
    tamanho = n;
    px = new int[n];
    for (int i = 0; i < n; i++){
        px[i] = i+1;
    }
}

Lista::~Lista(){
    delete[] px;
}

```

A função não recebe parâmetros e é executada sempre que um objeto será destruído.

Atributos da lista

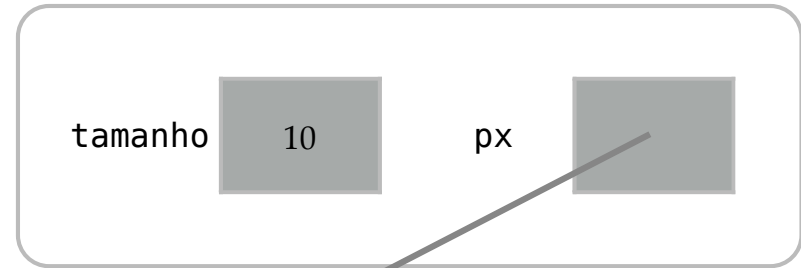


```
class Lista {  
    public:  
        Lista(int);  
        ~Lista();  
        void setElemento(int pos, int x) {px[pos-1] = x;}  
        int getElemento(int pos) {return px[pos-1];}  
    private:  
        int *px;  
        int tamanho;  
};  
  
Lista::Lista(int n){  
    tamanho = n;  
    px = new int[n];  
    for (int i = 0; i < n; i++){  
        px[i] = i+1;  
    }  
}  
  
Lista::~~Lista(){  
    delete[] px;  
}
```



Na implementação do destrutor pedemos dizer ao objeto para desalocar a memória apontada por `px` antes de excluir os atributos da lista.

Atributos da lista



```
class Lista {  
    public:  
        Lista(int);  
        ~Lista();  
        void setElemento(int pos, int x) {px[pos-1] = x;}  
        int getElemento(int pos) {return px[pos-1];}  
    private:  
        int *px;  
        int tamanho;  
};
```

```
Lista::Lista(int n){  
    tamanho = n;  
    px = new int[n];  
    for (int i = 0; i < n; i++){  
        px[i] = i+1;  
    }  
}
```

```
Lista::~~Lista(){  
    delete[] px;  
}
```



Apenas após a execução do destrutor é que os atributos da lista são deletados.