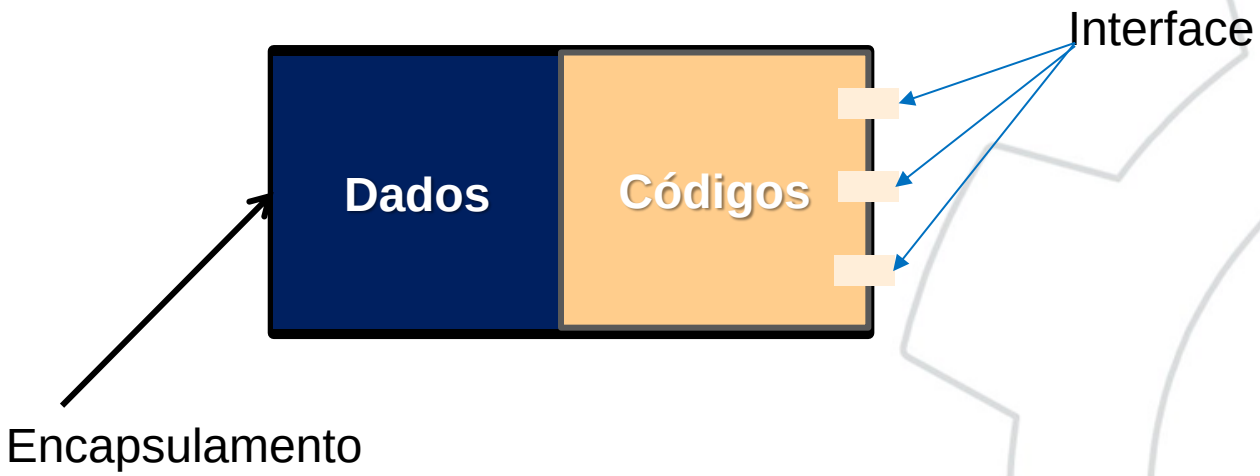
Aula 03:

Classes – Parte I

ECOP13A - Programação Orientada a Objetos
Prof. André Bernardi
andrebernardi@unifei.edu.br

Universidade Federal de Itajubá

CLASSES



As **classes** do C++ são uma ferramenta para a criação de **tipos** de dados, que podem ser utilizados com a mesma facilidade e conveniência dos tipos nativos.

Mas o que são tipos?

Um tipo é uma representação concreta de um conceito, de uma ideia, de uma noção. Por exemplo, o tipo nativo **float** do C++, com suas operações de +, -, *, etc. provê uma aproximação concreta e bastante satisfatória do conceito matemático de um **número real**.

Resumindo, portanto, tudo em uma única frase:

Uma **classe** é um **tipo** que é **definido pelo usuário**.

É necessário projetar novos tipos de dados para prover uma definição de conceitos que não possuam contrapartes diretas entre os tipos nativos.

Veja alguns exemplos:

- Nós poderíamos prover um novo tipo Motor em um programa que trate de automobilismo;
- Um tipo Personagem, em um jogo de RPG;
- Um tipo Paragrafo, em um editor de textos;
- Um tipo Proteina, em um programa científico.
- Etc., etc. e etc.

Traduza o **vocabulário do problema** para seu programa:

Um programa que provê tipos que se aproximam dos conceitos da aplicação tende a ser muito mais fácil de se entender, de projetar e de modificar. Um conjunto bem escolhido de tipos definidos pelo usuário, ou classes, também faz um programa mais conciso e passível de análise, além de menos propenso a erros.

A ideia fundamental da construção de novos tipos é a separação entre:

- **Detalhes específicos de implementação**;
  A disposição dos dados utilizados para armazenamento do objeto, como suas variáveis membro e implementação de funções de uso interno.

- **E as propriedades que são essenciais para sua correta utilização.**
  A lista completa de funções que realizam tarefas importantes e proveem acesso aos dados.

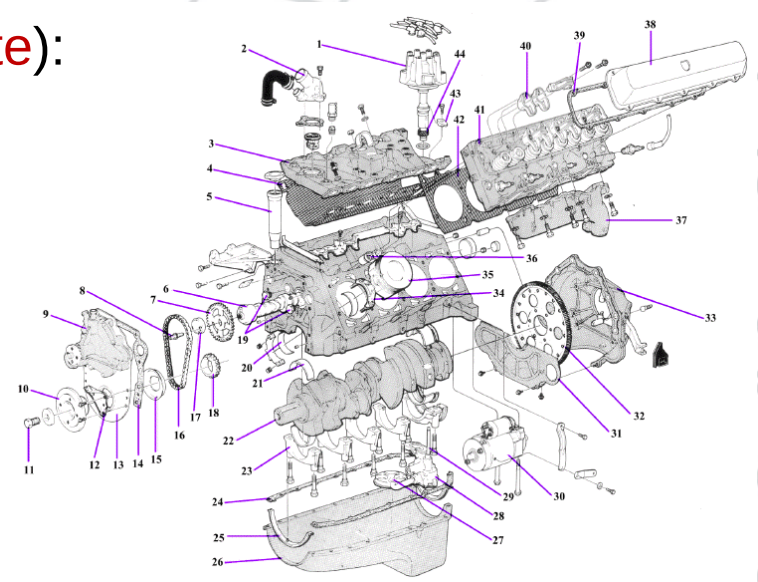
Lembre-se do mundo real: Não é necessário conhecer detalhes técnicos de um carro para dirigi-lo ou a configuração completa de um *smartphone* para realizar ligações e navegar na internet.

Temos algumas informações importantes:

- Uma classe é um **tipo definido pelo usuário**;
- Uma classe consiste em um conjunto de membros;
- Membros de dados (Atributos) e funções membro (Métodos).
- Possuem **construtores**, responsáveis pela sua inicialização, e **destrutores**, responsáveis pela finalização e limpeza;
- Membros, tanto de **dados** quanto **funções**, são acessados através da utilização do **.** **(ponto)** para objetos e **->** **(seta)** para ponteiros, de maneira semelhante às structs.
- Os **membros públicos** de uma classe (**public**) proveem sua interface, enquanto os **membros privados** proveem detalhes de implementação (**private**).
- Uma **struct** é como uma classe onde todos os membros são públicos. Utilize-as quando modelar um tipo que contenha apenas dados.

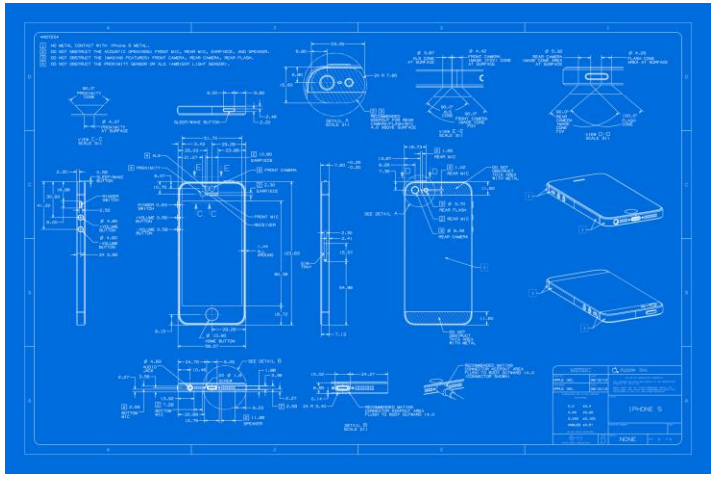
Interface pública de um carro (public)

Detalhes privados de projeto e implementação (private):



Interface pública de um smartphone (public)

Detalhes privados de projeto e implementação (private):



Uma descrição de classe será composta por três elementos:

- Declarações de membros de dados;
- Protótipos para funções membros;
- E definições das funções membros.

Existem várias maneiras de se organizar uma classe em arquivos: podemos descrever toda a classe em um único arquivo que também contenha a função *main*, e podemos também separar toda a descrição da classe em um arquivo único, separado, de cabeçalho (.h).

A maneira mais organizada é dividir a implementação da classe em dois arquivos, um deles com as declarações de função membro e protótipos de funções membro **(.h)** e outro com as implementações das funções **(.cpp)**. O programa principal ficaria ainda em um terceiro arquivo.

Repare que, desta maneira, teremos:

Um arquivo de “cabeçalho”, que contém apenas informações sobre os membros de dados e protótipos de funções. Funciona como uma interface, mostrando ao usuário o que pode ser utilizado.
Ex.: circle.h

Contém as **implementações** das funções cujo protótipo foram incluídos no arquivo de cabeçalho. As funções devem ser definidas dentro do escopo da classe, utilizando-se o operador **::**
Ex.: circle.cpp

Contém o **programa principal,** que apenas utiliza a classe definida nos dois arquivos anteriores, por exemplo, instanciando um objeto daquele tipo.
Ex.: principal.cpp

Compilando:

```
g++ -std=c++11 circle.cpp principal.cpp -o principal.exe
```

A estrutura de uma classe

A definição de uma classe sempre começa com a palavra reservada **class** e o nome da classe, que geralmente é iniciado com uma letra maiúscula.

A seguir, abre-se chaves, que delimitarão o conteúdo da definição da classe. Veja um exemplo:

```
class Driver {

        ... // definições da classe

};
```

Após o fechamento das chaves, é obrigatória a utilização de um ponto e vírgula.

A estrutura de uma classe

A definição de uma classe poderá conter uma ou mais seções, que definem a "**visibilidade**" ou "**nível de acesso**" de seus membros:

`public`: define que os membros ali declarados poderão ser acessados e utilizados em qualquer parte de um programa, bastando para isso, a "inclusão" de seu arquivo de cabeçalho através da diretiva *#include*.

`private`: define uma seção privada, que conterá os membros de dados e funções membro que não devem (e não podem) ser utilizadas de fora do escopo da classe (veremos sobre "*friends*" em breve). Funções privadas são chamadas de "funções utilitárias" e são utilizadas para "serviços internos" das classes.

`protected`: veremos em breve.

A estrutura de uma classe

Veja como ficariam as coisas:

```
class Driver {
        private:
                int age;
                char license_type;
                // Todos os dados que não poderão ser acessados
                // de fora da sua classe: encapsulamento.
        public:
                Driver();
                Driver(int, char);
                ~Driver();
                void print_profile();
                // Toda a interface pública da classe.
};
```

Construtores e Destrutor

Os construtores são identificados como funções membro cujos nomes são iguais ao nome da classe. Elas não possuem tipo de retorno. Ex.: Driver(), etc.

> Sua função é inicializar um objeto de sua classe sempre que ele for instanciado em um programa, cuidando para que os membros de dados contenham valores iniciais válidos e que recursos (como memória) sejam alocados propriamente.

Uma classe pode ter mais de um construtor, contanto que suas listas de parâmetros sejam diferentes. Um construtor pode ter uma lista de parâmetros vazia. Os construtores podem fornecer valores *default* para um ou mais de seus parâmetros.

Construtores e Destrutor

Caso uma classe não inclua explicitamente um construtor, o compilador oferecerá um construtor padrão (*default*), que nada mais é que um construtor sem parâmetros.

> O construtor padrão fornecido pelo compilador não inicializa nenhum membros de dados com valores específicos: caso sejam de tipos nativos, nada é realizado (valor = lixo), caso sejam objetos de outras classes, ele chama o construtor padrão de cada uma destas classes para garantir uma inicialização própria.

Você pode definir um construtor padrão explicitamente, ou seja, sem parâmetros. Ele chamará o construtor padrão para cada membro de dados de outra classe e realizará a inicialização adicional provida por você.

Veja um exemplo:

```
// inicializa cada atributo com um valor padrão
Driver:: Driver()
{
        age = 20;
        license_type = ‘b’;
}

// inicializa com valores cedidos pelo usuário
Driver::Driver(int a, char lt)
{
        age = a;
        license_type = lt;
}
```

Se você definir um construtor com argumentos, C++ não criará um construtor *default* implicitamente. **Dica**: Seja explícito sempre.

E quando será chamado um construtor?

No ato da declaração de um objeto do tipo da classe especificada. Veja:

```
// Incluindo cabeçalho da classe Driver
#include “driver.h”

int main()
{
        // Definindo um motorista padrão, de 20 anos e carteira ‘b’
        // Construtor default é chamado
        Driver d1;

        //  Define um motorista específico, de 50 anos e carteira ‘d’
        // Chama construtor com parâmetros (int, char)
        Driver d2(50,’d’);
}
```

Construtores e Destrutor

O destrutor, por sua vez, possui sempre o nome da classe precedido por um ‘til’ (~) e sua lista de parâmetros é sempre vazia. Não possui tipo de retorno. Ex.: ~Driver(), etc.

O destrutor de um objeto é chamado **implicitamente** sempre que um objeto daquela classe precisa ser destruído, por exemplo, quando a execução deixa o escopo a que ele pertence. Realiza uma faxina de finalização antes que o sistema recupere a memória utilizada pelo objeto. Uma memória que tenha sido alocada dinamicamente em um construtor, **deve** ser desalocada no destrutor.

Não é permitido ter mais de um destrutor por classe.

Construtores e Destrutor

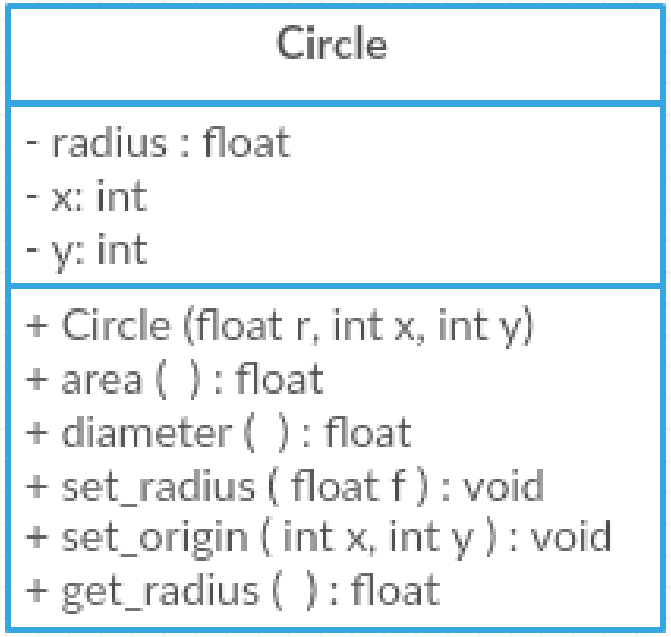
Caso você não inclua um destrutor, o compilador cria um destrutor vazio.

São importantes para finalização de objetos que utilizam recursos do sistema, como memória alocada dinamicamente (ponteiros), arquivos, etc. Veja um exemplo:

```
Driver::~Driver()
{
    // Caso não seja realizado aqui, o processo de liberação
    // da memória utilizada em destinations nunca será feito
    delete[] destinations;
}
```

Exemplo

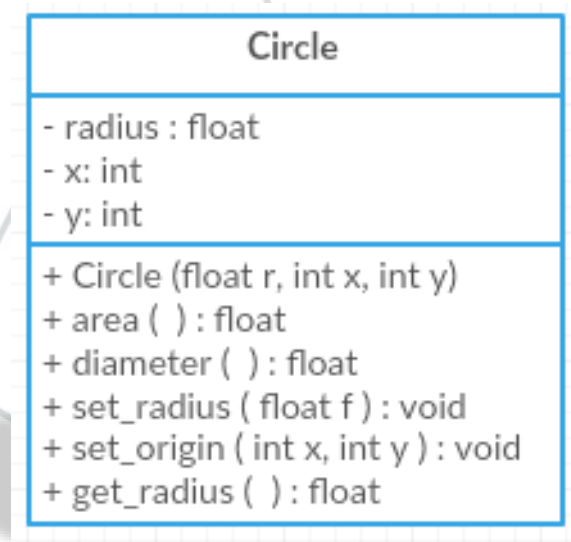
Vamos criar um tipo de dados "**Círculo**", que modela as características e comportamentos próprios deste tipo de figura geométrica. Utilizaremos dois arquivos para definição da classe: um de cabeçalho (.h), e outro de implementação (.cpp).



A UML (ou *Unified Modeling Language*) permite a representação gráfica de classes, de uma maneira **independente de linguagens de programação**. A partir de agora, também utilizaremos este tipo de notação, mas de maneira simplificada.

É composta por três seções: **Nome da classe**, seção com **membros de dados** e seção com **funções membro**.

```cpp
// Definição da classe Circle (arquivo: circle.h)
// Funções-membro serão definidas em circle.cpp

#ifndef CIRCLE_H
#define CIRCLE_H

class Circle {
    private:
        float radius;
        int x, y;
    public:

        // Construtores e Destrutores
        Circle();
        Circle(float, int = 0, int = 0);
        ~Circle() {}

        // Funções de interface
        float area();
        float diameter();
        void set_radius(float);
        void set_origin(int, int);
        float get_radius();
};

#endif
```



(-) membro private
(+) membro public

Include Guards

As diretivas de compilação #ifndef, #define e #endif são muito importantes para o funcionamento normal do programa. Elas querem dizer, exatamente: “Compilador, se ainda não definiu CLASSNAME_H, defina. Sua definição vai do #define ao #endif”.

Com esta ação, o programador previne que um determinado arquivo de cabeçalho seja incluído mais de uma vez, o que poderia gerar erros de compilação, como redefinição de identificadores (tipos, enum, variáveis), por exemplo.

Ao utilizá-los, garantimos que cada arquivo de cabeçalho será incluído uma única vez.


```cpp
// Definições das funções membro da classe Circle
// arquivo (circle.cpp)

#include "circle.h"

// Construtor padrão: Não recebe parâmetros
// Inicializa um objeto default, com valores 0.
Circle::Circle()
{
    x = y = 0;
    radius = 0;
}

// Construtor
// Recebe parâmetros de inicialização
Circle::Circle(float r, int vx, int vy)
{
    radius = r;
    x = vx;
    y = vy;
}
```

```cpp
// Calcula a área do círculo
float Circle::area()
{
    return 3.141593 * radius * radius;
}

// Calcula o diâmetro do círculo
float Circle::diameter()
{
    return 2 * radius;
}

// Dá acesso ao usuário, para modificar
// valor de radius
void Circle::set_radius(float r)
{
    radius = r;
}

// Dá acesso ao usuário, para modificar
// valor da coordenada de origem
void Circle::set_origin(int x, int y)
{
    this->x = x;
    this->y = y;
}

// Dá acesso ao usuário, para obter
// valor de radius
float Circle::get_radius()
{
    return radius;
}
```

Getters e Setters – *Tome Cuidado...*

Sua classe pode conter funções membro públicas que concedam acesso a membros de dados privados, tanto para **consultar** um dado (***get***) quanto para **modificar** seu valor (***set***).

> No entanto, seja cauteloso. Funções ***set***, como set_radius, podem ferir o conceito de encapsulamento da programação orientada a objetos, fazendo com que a lógica de seu programa fique em outro lugar, fora da própria classe.

Imagine uma função ***set_speed*** para uma classe automóvel. No mundo real, não seria natural estabelecer uma determinada velocidade instantaneamente. Seria preciso que a velocidade começasse em zero e um comportamento de aceleração levasse até a velocidade desejada.


```cpp
// programa principal para utilização da
// classe Circle

#include <iostream>
#include "circle.h" // incluindo classe

using namespace std;

int main()
{
    Circle c1;
    Circle c2(10, 10, 5);
    Circle c3(12.5);

    cout << "Diametro de cada circulo: \n";
    cout << "c1 = " << c1.diameter() << "\n";
    cout << "c2 = " << c2.diameter() << "\n";
    cout << "c3 = " << c3.diameter() << "\n";

    cout << "Area de cada circulo: \n";
    cout << "c1 = " << c1.area() << "\n";
    cout << "c2 = " << c2.area() << "\n";
    cout << "c3 = " << c3.area() << "\n";
}
```

```
D:\Disciplinas\ECOP03 - POO\Aulas\3 - Classes - Parte I\code>g++ -std=c++11 circle.cpp principal.cpp -o principal.exe

D:\Disciplinas\ECOP03 - POO\Aulas\3 - Classes - Parte I\code>principal
Diametro de cada circulo:
c1 = 0
c2 = 62.8319
c3 = 78.5398
Area de cada circulo:
c1 = 0
c2 = 98.6961
c3 = 123.37
```

Retorno de Referência

Além dos *Getters* e *Setters*, existe uma outra armadilha bastante sutil que pode ferir o encapsulamento da classe: o retorno, em uma função membro, de uma referência para um membro de dados. Veja um exemplo:

```
int& Time::bad_set_hour(int hh)
{
        hour = ((hh < 24 && hh >= 0) ? hh : 0);
        return hour;
}
```

Ao utilizar esta técnica, mais uma vez nosso código deixa de ser protegido pelos limites da classe, de maneira que o usuário poderá, por exemplo, inicializar o membro *hour* com um valor inválido, através da referencia retornada.

Atribuição com cópia membro a membro default

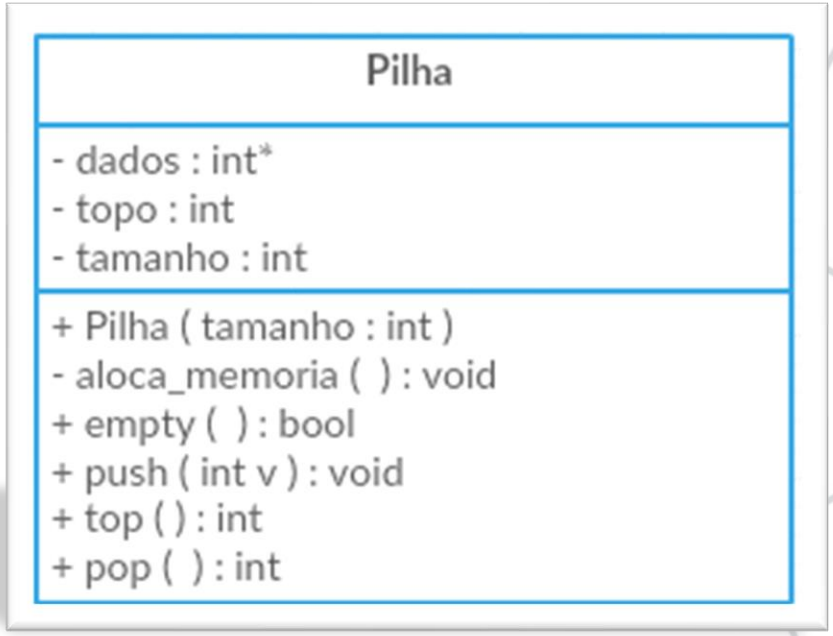
O operador de atribuição (=) pode ser utilizado para atribuir um objeto para outro do mesmo tipo. Como padrão, esta atribuição é realizada membro a membro. Veja um exemplo:

```
int main()
{
    Circle c1(12, 0, 0);
    Circle c2;

    c2 = c1;                // c2.radius = 12, c2.x = 0, c2.y = 0
}
```

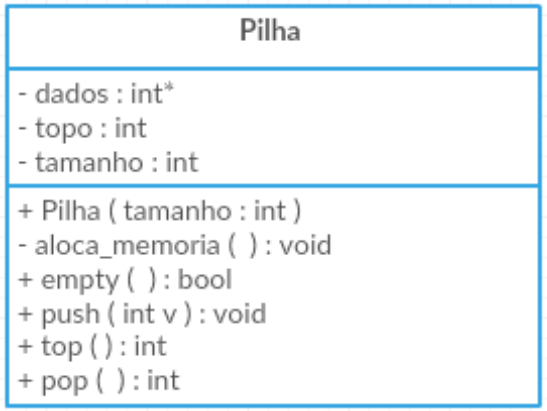
Tome cuidado caso sua classe possua membros de dados que são ponteiros alocados dinamicamente.

Exemplo II

Vamos criar um tipo de dados “Pilha”, que modela as características e comportamentos próprios deste tipo de estrutura de dados (*Last-in, First-out*).




```
// Interface da classe Pilha (stack.h)
// Definições de funções em stack.cpp

#ifndef PILHA_H
#define PILHA_H

class Pilha {

    private:
        int *dados;
        int topo;
        int tamanho;

        // função utiltária (ou auxiliar)
        void aloca_memoria();

    public:
        Pilha();
        Pilha(int);
        ~Pilha();

        bool empty(); // verifica se vazia (Função de acesso)
        void push(int); // insere no topo
        int top(); // visualiza topo
        int pop(); // retira do topo
};

#endif
```




```cpp
// definições de funções membro da
// classe Pilha (stack.cpp)

#include "stack.h"
#include <iostream>
using namespace std;

// Função privada de alocação de memória
// Não faz sentido fazer parte da interface pública
void Pilha::aloca_memoria()
{
    if(tamanho > 0)
        dados = new int[tamanho];
    else
        dados = nullptr;
}

// Construtor Padrão
Pilha::Pilha()
{
    // tamanho padrão da pilha
    tamanho = 100;
    topo = -1;
    aloca_memoria();
}
```

Repare que a função membro *aloca_memoria* nunca deveria ser utilizada por um usuário da classe. Por isso a definimos como uma função utilitária, na seção *private*.

```cpp
// Construtor com parâmetros
Pilha::Pilha(int t)
{
    if(t < 0) t = 0; // verificação de segurança
    tamanho = t;
    topo = -1;
    aloca_memoria();
    cout << "construindo..." << endl;
}

// Destrutor:
// responsável pela limpeza da classe e
// desalocação de recursos.
Pilha::~Pilha()
{
    if(dados != nullptr)
        delete[] dados;
    cout << "destruindo..." << endl;
}

// verifica se pilha está vazia
bool Pilha::empty()
{
    return (topo == -1 ? true : false);
}
```

Existe um grande cuidado para que a pilha nunca entre em um estado inexistente, com tamanho negativo, por exemplo.

Além disso, repare como o destrutor realiza a limpeza do objeto antes de devolvê-lo ao sistema.


```cpp
// se possível, insere novo item na pilha
void Pilha::push(int v)
{
    // verificações de segurança
    if(dados != nullptr && topo != tamanho-1)
    {
        // inserção
        topo++;
        dados[topo] = v;
    }
}

// retorna elemento no topo da pilha
int Pilha::top()
{
    if(!empty())
        return dados[topo];
}

// retira elemento do topo da pilha
int Pilha::pop()
{
    // verifica se há elementos e retorna dado
    if(!empty())
        return dados[topo--];
}
```

Os métodos sempre realizam verificações de segurança em cada operação, certificando que o usuário nunca cause erros no processo de utilização.

```cpp
#include <iostream>
#include "stack.h"

using namespace std;

int main()
{
    Pilha p(50);

    cout << "Pilha esta vazia? ";

    if(p.empty()) cout << "SIM\n";
    else cout << "NAO\n";

    cout << "Inserindo elementos... " << "\n";
    p.push(10);
    p.push(20);
    p.push(30);

    cout << "Que esta no topo? " << p.top() << "\n";
    cout << "Retirando elemento... " << p.pop() << "\n";
    cout << "Retirando elemento... " << p.pop() << "\n";
    cout << "Que esta no topo? " << p.top() << "\n";

}
```

Exemplo de programa principal, que inclui a classe Pilha e a utiliza.

Construtor de inicialização ou cópia

O operador de atribuição (=) pode ser utilizado inicializar um objeto com valores de um outro previamente existente. Como padrão, esta atribuição é realizada membro a membro.

Para que possamos controlar como um objeto é copiado para o outro devemos prover um construtor de cópia.

```
class CPilha
{
    public:
            CPilha( const CPilha& );    //construtor de cópia
    ...
} ;
```

Quando passamos um objeto como parâmetro para uma função por valor, o construtor de cópia também é chamado.

Construtor de inicialização ou cópia

```
CPilha p1;          // declarar um objeto pilha default
CPilha p2 (20);     // declarar um objeto pilha personalizado
CPilha p3 = p1;     // inicializar p3 com valores de p1
```

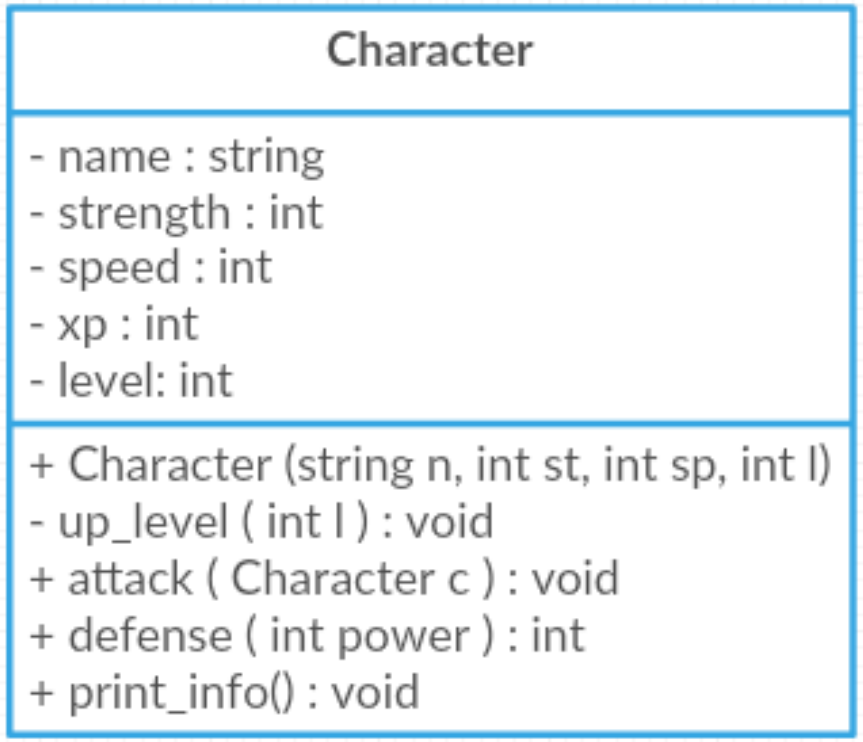
A terceira declaração acima, utiliza o construtor de cópia.

Observe que devemos implementá-lo necessariamente neste caso, pois a classe CPilha utiliza alocação dinâmica em seus membros.

Caso não seja provido o construtor de cópia o programa poderia apresentar bugs inesperados, como por exemplo a criação de um objeto "*siamês*", onde os dois objetos apontariam para a mesma posição de memória onde os dados da pilha estão armazenados.

Exercício

Crie um tipo de dados “Character”, que modela as características e comportamentos próprios de um personagem de um jogo de RPG. A seguir, crie um programa principal que realiza uma batalha entre dois desses personagens.



Referências

- https://cplusplus.com/reference/

- Notas de aula da disciplina Programação Orientada a Objetos, Prof. André Bernardi, Prof. João Paulo Reus Rodrigues Leite.

Construtor de inicialização ou cópia

```
CPilha p1;          // declarar um objeto pilha default
CPilha p2 (20);     // declarar um objeto pilha personalizado
CPilha p3 = p1;     // inicializar p3 com valores de p1
```

A terceira declaração acima, utiliza o construtor de cópia.

Observe que devemos implementá-lo necessariamente neste caso, pois a classe CPilha utiliza alocação dinâmica em seus membros.

Caso não seja provido o construtor de cópia o programa poderia apresentar bugs inesperados, como por exemplo a criação de um objeto "*siamês*", onde os dois objetos apontariam para a mesma posição de memória onde os dados da pilha estão armazenados.

```
// Fig. 15.14: fig15_14.cpp
// Demonstrating C++ Standard Library class template vector.
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

void outputVector( const vector< int > & ); // display the vector
void inputVector( vector< int > & ); // input values into the vector

int main()
{
   vector< int > integers1( 7 ); // 7-element vector< int >
   vector< int > integers2( 10 ); // 10-element vector< int >

   // print integers1 size and contents
   cout << "Size of vector integers1 is " << integers1.size()
      << "\nvector after initialization:" << endl;
   outputVector( integers1 );

   // print integers2 size and contents
   cout << "\nSize of vector integers2 is " << integers2.size()
      << "\nvector after initialization:" << endl;
   outputVector( integers2 );
```